EXPERT INSIGHT

Terraform Cookbook

Provision, run, and scale cloud architecture with real-world examples using Terraform

Foreword by:
Armon Dadgar
CTO and Co-Founder, HashiCorp

Second Edition

Mikael Krief

packt

Terraform Cookbook

Second Edition

Provision, run, and scale cloud architecture with real-world examples using Terraform

Mikael Krief

packt

BIRMINGHAM—MUMBAI

Terraform Cookbook

Second Edition

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Rahul Nair

Acquisition Editor – Peer Reviews: Saby Dsilva

Project Editor: Parvathy Nair

Content Development Editors: Georgia Daisy van der Post, Matthew Davies

Copy Editor: Safis Editing

Technical Editor: Kushal Sharma

Proofreader: Safis Editing

Indexer: Rekha Nair

Presentation Designer: Ganesh Bhadwalkar

Developer Relations Marketing Executive: Priyadarshini Sharma

First published: October 2020

Second edition: August 2023

Production reference: 1250823

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul’s Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80461-642-0

`www.packt.com`

I would like to dedicate this book to my wife and children, who are my source of happiness.

– Mikael Krief

Foreword

It was 2013 when Mitchell and I first conceived of Terraform, 10 years before the publication of the second edition of the *Terraform Cookbook*. At the time, we felt that the growing complexity of cloud environments required a radically different approach to management, and that Infrastructure as Code would play a critical enabling role. Since then, Terraform has grown to have a massive ecosystem of thousands of integrations, tens of thousands of modules, hundreds of thousands of users, and more than one billion downloads.

When we first built Terraform, we were focused on the workflow and how we wanted it to be used. It was important that this be consistent and simple, regardless of what types of resource we were managing, whether public clouds, private clouds, network devices, or SaaS services. Given the complexity of modern cloud environments, we had to provide confidence to end users and ensure they were never surprised. We also knew there was an almost infinite surface area of integration, so it had to be easy to create plugins to extend Terraform.

Terraform today delivers on all those goals. There are multiple ways to author Terraform, whether with **HashiCorp Configuration Language (HCL)**, **JavaScript Object Notation (JSON)**, or through programming languages such as TypeScript or Python using the Terraform CDK. Terraform is easily extensible through providers, which enables thousands of integrations across low-level hardware, cloud services, and SaaS. The rich ability to plan changes provides operators with the confidence they need around changes, which is why it's used by thousands of organizations to manage their production environments.

I was excited when Mikael Krief wrote the first version of the *Terraform Cookbook* to provide a practical guide for new users to learn the tool and apply it in a number of real-world situations. With the second edition, Mikael is providing an important refresh that covers many of the updates to the core Terraform product, which has evolved rapidly over the last few years. He also brings in many of the best practices that have evolved as the community has spent more time figuring out how to manage infrastructure at scale.

This book starts with a very gentle introduction, including how to download and set up Terraform, and is perfect for users who are just getting started. From there, it introduces the basics of authoring Terraform code and using key features of Terraform. These lessons are brought together through more complex examples that present real-world use cases to help readers go from the basics of the tool to the practical usage of it.

For new users just getting started with Terraform or Infrastructure as Code, this book will provide a valuable way to get started quickly. For users who haven't used Terraform in a few years, this will provide an updated view of the new features and patterns that have emerged in the last few years.

I hope you enjoy the book!

Armon Dadgar
CTO and Co-Founder, HashiCorp

Contributors

About the author

Mikael Krief is a DevOps engineer who lives in France. He believes that Infrastructure as Code is a fundamental practice in DevOps culture. He is therefore interested in HashiCorp products and specializes in the use of Terraform. Mikael loves to share his passion through various communities, such as the HashiCorp User Groups. Over the years, he has contributed to many public projects, written various blog posts, published several books, and spoken at leading conferences. For his contributions and passion, he has been nominated and selected as a HashiCorp Ambassador since 2019, and he has been awarded the Microsoft Most Valuable Professional (MVP) award for 8 years.

I would like to extend my thanks to my family for accepting that I needed to work long hours on this book during family time. I would like to thank Meeta Rajani for giving me the opportunity to write this second edition, which was a very enriching experience. Special thanks to Parvathy Nair, Georgia Daisy van der Post, and Radek Simko for their valuable input and time reviewing this book and to the entire Packt team for their support during the course of writing this book.

About the reviewer

Jack Lee is a Microsoft MVP and an Azure Certified Solutions Architect with a passion for software development, the cloud, and DevOps innovation. He is an active Microsoft Tech Community contributor and has presented at various user groups and conferences, including the Global Azure Bootcamp at Microsoft Canada. Jack is an experienced mentor and judge at hackathons and is also the president of a user group that focuses on Azure, DevOps, and software development. He is the co-author of *Azure for Architects*, *Azure Strategy and Implementation Guide*, and *Cloud Analytics with Microsoft Azure* from Packt Publishing. You can follow Jack on Twitter at `@jlee_consulting`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

Table of Contents

Preface xxxv

Chapter 1: Setting Up the Terraform Environment 1

Technical requirements 2

Overviewing Terraform best practices 2

Downloading and installing Terraform on Windows manually 4

Getting ready • 4

How to do it... • 4

How it works... • 9

Installing Terraform using Chocolatey on Windows 11

Getting ready • 11

How to do it... • 12

How it works... • 12

There's more... • 13

See also • 13

Installing Terraform on Linux using the APT package manager 13

Getting ready • 13

How to do it... • 13

How it works... • 14

See also • 14

Installing Terraform using a script on Linux 14

Getting ready • 14

How to do it… • 15

How it works… • 15

There's more… • 16

See also • 17

Executing Terraform in a Docker container 17

Getting ready • 17

How to do it… • 18

How it works… • 20

There's more… • 20

See also • 20

Switching between multiple Terraform versions 21

Getting ready • 22

How to do it… • 22

How it works… • 25

There's more… • 25

See also • 27

Upgrading Terraform providers 27

Getting ready • 27

How to do it… • 28

How it works… • 31

There's more… • 31

See also • 32

Chapter 2: Writing Terraform Configurations 33

Technical requirements 34

Configuring Terraform and the provider version to use 34

Getting ready • 35

How to do it… • 37

How it works… • 37

There's more… • 39

See also • 40

Adding alias to a provider to use multiple instances of the same provider 40

Getting ready • 40

How to do it… • 41

How it works… • 42

See also • 42

Manipulating variables 43

Getting ready • 43

How to do it… • 43

How it works… • 44

There's more… • 45

See also • 46

Keeping sensitive variables safe 47

Getting ready • 47

How to do it… • 47

How it works… • 48

There's more… • 48

See also • 49

Using local variables for custom functions 49

Getting ready • 49

How to do it… • 50

How it works… • 51

There's more… • 51

See also • 51

Using outputs to expose Terraform provisioned data 52

Getting ready • 52

How to do it… • 53

How it works… • 53

There's more… • 53

See also • 54

Calling Terraform's built-in functions 54

Getting ready • 54

How to do it… • 55
How it works… • 55
See also • 56
Using YAML files in Terraform configuration 57
Getting ready • 57
How to do it… • 57
How it works… • 59
There's more… • 59
See also • 59
Writing conditional expressions 59
Getting ready • 60
How to do it… • 60
How it works… • 60
There's more… • 61
See also • 61
Generating passwords with Terraform 61
Getting ready • 62
How to do it… • 62
How it works… • 63
See also • 63
Managing Terraform resource dependencies 64
Getting ready • 64
How to do it… • 65
How it works… • 65
There's more… • 66
See also • 66
Adding custom pre and postconditions 66
Getting ready • 67
How to do it… • 67
How it works… • 68
See also • 70

Using checks for infrastructure validation 70

Getting ready • 70

How to do it… • 71

How it works… • 71

There’s more… • 72

See also • 72

Chapter 3: Scaling Your Infrastructure with Terraform 75

Technical requirements 76

Provisioning infrastructure in multiple environments 76

Getting ready • 76

How to do it… • 77

How it works… • 78

See also • 81

Provisioning multiple resources with the count meta-argument 81

Getting ready • 81

How to do it… • 82

How it works… • 82

There’s more… • 84

See also • 85

Using maps 85

Getting ready • 86

How to do it… • 86

How it works… • 87

There’s more… • 88

See also • 89

Looping over a map of objects 89

Getting ready • 90

How to do it… • 90

How it works… • 91

There’s more… • 92

See also • 93

Generating multiple blocks with the dynamic block 93

Getting ready • 94

How to do it... • 94

How it works... • 96

There's more... • 97

See also • 97

Filtering maps 97

Getting ready • 98

How to do it... • 98

How it works... • 99

There's more... • 100

See also • 100

Chapter 4: Using Terraform with External Data 103

Technical requirements 104

Obtaining external data with data sources 104

Getting ready • 105

How to do it... • 105

How it works... • 105

There's more... • 106

See also • 107

Querying external data with Terraform 107

Getting ready • 108

How to do it... • 108

How it works... • 109

There's more... • 111

See also • 111

Manipulating local files with Terraform 111

Getting ready • 112

How to do it... • 112

How it works… • 113

There's more… • 113

See also • 114

Executing local programs with Terraform 114

Getting ready • 114

How to do it… • 114

How it works… • 115

There's more… • 116

See also • 117

Executing shell scripts using the Terraform Shell provider 117

Getting ready • 117

How to do it… • 117

How it works… • 119

There's more… • 121

See also • 121

Chapter 5: Managing Terraform State 123

Using the local Terraform state 124

Getting ready • 124

How to do it… • 124

How it works… • 125

There's more… • 125

See also • 127

Managing resources in Terraform state 127

Getting ready • 127

How to do it… • 128

Displaying the content of state • 128

Listing Terraform resource names within state • 130

Showing detailed resource properties in state • 130

Deleting resources from state • 131

There's more… • 132

See also • 132

Synchronizing Terraform state 132

Getting ready • 133

How to do it… • 134

How it works… • 135

There's more… • 137

See also • 137

Importing existing resources 137

Getting ready • 138

How to do it… • 139

How it works… • 140

There's more… • 140

See also • 143

Using external resources from other Terraform state files 143

Getting ready • 144

How to do it… • 145

How it works… • 145

There's more… • 146

See also • 146

Refactoring resources in configuration 146

Getting ready • 147

How to do it… • 149

How it works… • 150

There's more… • 153

See also • 154

Chapter 6: Applying a Basic Terraform Workflow 157

Technical requirements 158

Keeping your Terraform configuration clean 158

Getting ready • 159

How to do it… • 159

How it works... • 159

There's more... • 160

See also • 162

Validating the code syntax 162

Getting ready • 162

How to do it... • 163

How it works... • 164

There's more... • 164

See also • 166

Destroying infrastructure resources 166

Getting ready • 166

How to do it... • 167

How it works... • 167

There's more... • 167

See also • 168

Displaying a list of providers used in a configuration 169

Getting ready • 169

How to do it... • 170

How it works... • 170

There's more... • 170

See also • 171

Generating one Terraform lock file with Windows and Linux compatibility 171

Getting ready • 172

How to do it... • 173

How it works... • 173

See also • 174

Copying a Terraform module configuration 175

Getting ready • 175

How to do it... • 175

How it works... • 175

There's more... • 176

See also • 176

Using workspaces to manage environments 177

Getting ready • 177

How to do it… • 177

How it works… • 178

There's more… • 180

See also • 181

Exporting the output in JSON 181

Getting ready • 181

How to do it… • 182

How it works… • 182

There's more… • 184

See also • 185

Tainting resources 185

Getting ready • 185

How to do it… • 185

How it works… • 186

There's more… • 187

See also • 188

Generating the dependency graph 188

Getting ready • 189

How to do it… • 189

How it works… • 189

See also • 190

Using different Terraform configuration directories 190

Getting ready • 191

How to do it… • 192

How it works… • 192

There's more… • 192

See also • 193

Testing and evaluating a Terraform expression 193

Getting ready • 193

How to do it… • 194

How it works… • 194

There's more… • 195

See also • 196

Debugging the Terraform execution 196

Getting ready • 196

How to do it… • 196

How it works… • 197

There's more… • 198

See also • 198

Chapter 7: Sharing Terraform Configuration with Modules 199

Technical requirements 200

Creating a Terraform module and using it locally 200

Getting ready • 201

How to do it… • 201

How it works… • 204

There's more… • 204

See also • 206

Provisioning multiple instances of a Terraform module 206

Getting ready • 206

How to do it… • 207

How it works… • 208

There's more… • 208

See also • 209

Using modules from the public registry 209

Getting ready • 209

How to do it… • 210

How it works… • 211

There's more… • 212

See also • 213

Sharing a Terraform module in the public registry using GitHub 213

Getting ready • 213

How to do it… • 213

How it works… • 216

There's more… • 217

See also • 218

Using another file inside a custom module 218

Getting ready • 218

How to do it… • 219

How it works… • 219

There's more… • 220

See also • 221

Using the Terraform module generator 221

Getting ready • 221

How to do it… • 222

How it works… • 222

There's more… • 224

See also • 224

Generating module documentation 225

Getting ready • 225

How to do it… • 226

How it works… • 226

There's more… • 228

See also • 229

Using a private Git repository for sharing a Terraform module 229

Getting ready • 230

How to do it… • 231

How it works… • 233

There's more… • 234

See also • 235

Applying a Terrafile pattern for using modules 235

Getting ready • 236

How to do it… • 236

How it works… • 238

There's more… • 238

See also • 239

Chapter 8: Provisioning Azure Infrastructure with Terraform 241

Technical requirements 242

Using Terraform in Azure Cloud Shell 242

Getting ready • 243

How to do it… • 243

How it works… • 244

There's more… • 245

See also • 246

Protecting the Azure credential provider 246

Getting ready • 247

How to do it… • 247

How it works… • 249

There's more… • 250

See also • 251

Protecting the state file in the Azure remote backend 251

Getting ready • 252

How to do it… • 252

How it works… • 253

There's more… • 254

See also • 256

Executing ARM templates in Terraform 256

Getting ready • 257

How to do it… • 257

How it works... • 259

There's more... • 260

See also • 260

Executing Azure CLI commands in Terraform 261

Getting ready • 261

How to do it... • 262

How it works... • 263

There's more... • 263

See also • 264

Using Azure Key Vault with Terraform to protect secrets 265

Getting ready • 265

How to do it... • 266

How it works... • 268

There's more... • 269

See also • 271

Provisioning and configuring an Azure VM with Terraform 271

Getting ready • 271

How to do it... • 272

How it works... • 275

There's more... • 276

See also • 277

Building Azure serverless infrastructure with Terraform 277

Getting ready • 277

How to do it... • 278

How it works... • 279

There's more... • 279

See also • 280

Generating a Terraform configuration for existing Azure infrastructure 280

Getting ready • 281

How to do it... • 283

How it works... • 284

There's more... • 286

See also • 287

Enabling optional Azure features **287**

Getting ready • 287

How to do it... • 288

How it works... • 288

There's more... • 288

Estimating Azure cost of infrastructure using Infracost **288**

Getting ready • 289

How to do it... • 291

How it works... • 293

There's more... • 293

See also • 294

Using the AzApi Terraform provider **294**

Getting ready • 295

How to do it... • 295

How it works... • 297

There's more... • 298

See also • 299

Chapter 9: Provisioning Simple AWS and GCP Infrastructure Using Terraform **301**

Technical requirements **302**

Getting started using Terraform for AWS **302**

Getting ready • 302

How to do it... • 304

How it works... • 305

There's more... • 306

See also • 307

Using the S3 backend in AWS **307**

Getting ready • 307

How to do it... • 310

How it works... • 310

There's more... • 311

See also • 311

Getting started using Terraform for GCP 312

Getting ready • 312

How to do it... • 316

How it works... • 317

There's more... • 319

See also • 319

Using a GCS backend in GCP 319

Getting ready • 319

How to do it... • 322

How it works... • 322

See also • 323

Executing Terraform in GCP Cloud Shell 323

Getting ready • 324

How to do it... • 324

How it works... • 327

There's more... • 327

See also • 328

Chapter 10: Using Terraform for Docker and Kubernetes Deployment 329

Technical requirements 330

Creating a Docker container using Terraform 331

Getting ready • 331

How to do it... • 331

How it works... • 332

There's more... • 334

See also • 334

Deploying Kubernetes resources using Terraform 334

Getting ready • 335

How to do it… • 335

How it works… • 338

There's more… • 339

See also • 340

Deploying a Helm chart in Kubernetes using Terraform 340

Getting ready • 340

How to do it… • 341

How it works… • 342

There's more… • 343

See also • 344

Using a Kubernetes controller as a Terraform reconciliation loop 344

Getting ready • 345

How to do it… • 347

How it works… • 348

There's more… • 350

See also • 351

Chapter 11: Running Test and Compliance Security on Terraform Configuration 353

Technical requirements 355

Using PowerShell's Pester framework to perform Terraform testing 355

Getting ready • 356

How to do it… • 357

How it works… • 360

There's more… • 361

See also • 362

Testing the Terraform configuration using Python 362

Getting ready • 362

How to do it… • 363

How it works... • 365

There's more... • 366

See also • 366

Using OPA to check the Terraform configuration 366

Getting ready • 367

How to do it... • 369

How it works... • 371

There's more... • 372

See also • 373

Using tfsec to analyze the compliance of Terraform configuration 373

Getting ready • 374

How to do it... • 375

How it works... • 376

There's more... • 376

See also • 377

Applying Terraform compliance using terraform-compliance 377

Getting ready • 377

How to do it... • 378

How it works... • 380

There's more... • 382

See also • 382

Testing Terraform module code with Terratest 382

Getting ready • 382

How to do it... • 383

How it works... • 386

There's more... • 387

See also • 388

Testing the Terraform configuration using Kitchen-Terraform 388

Getting ready • 388

How to do it... • 389

How it works... • 392

There's more... • 394

See also • 394

Using the new integrated Terraform module integration test 395

Getting ready • 395

How to do it... • 396

How it works... • 398

There's more... • 399

See also • 400

Chapter 12: Deep-Diving into Terraform 401

Technical requirements 402

Preventing resources from being destroyed 402

Getting ready • 402

How to do it... • 403

How it works... • 404

There's more... • 404

See also • 406

Ignoring manual changes 406

Getting ready • 407

How to do it... • 408

How it works... • 409

There's more... • 409

See also • 410

Using Terraform's templating feature 410

Getting ready • 410

How to do it... • 412

How it works... • 413

There's more... • 414

See also • 414

Zero-downtime deployment with Terraform 415

Getting ready • 415

How to do it... • 416

How it works... • 417

There's more... • 417

See also • 418

Managing Terraform configuration dependencies using Terragrunt 418

Getting ready • 418

How to do it... • 420

How it works... • 420

There's more... • 421

See also • 421

Using Terragrunt as a wrapper for Terraform 422

Getting ready • 422

How to do it... • 423

How it works... • 424

See also • 424

Generating a self-signed SSL certificate using Terraform 425

Getting ready • 425

How to do it... • 425

How it works... • 426

There's more... • 426

See also • 428

Checking the configuration before committing code using Git hooks 429

Getting ready • 429

How to do it... • 430

How it works... • 431

There's more... • 432

See also • 432

Visualizing Terraform resource dependencies with Rover 433

Getting ready • 433

How to do it... • 433

How it works... • 434

See also • 435

Using the Terraform CDK for developers 435

Getting ready • 436

How to do it... • 437

How it works... • 441

There's more... • 443

See also • 444

Chapter 13: Automating Terraform Execution in a CI/CD Pipeline 445

Running Terraform in automation mode 446

Getting ready • 446

How to do it... • 447

How it works... • 448

There's more... • 448

See also • 448

Displaying a summary of the execution of terraform plan 449

Getting ready • 449

How to do it... • 449

There's more... • 451

See also • 451

Building CI/CD pipelines to apply Terraform configurations in Azure Pipelines 451

Getting ready • 452

How to do it... • 454

How it works... • 458

There's more... • 459

See also • 460

Automating Terraform execution in GitHub Actions 460

Getting ready • 461

How to do it... • 461

How it works... • 463

There's more... • 463

See also • 465

Working with workspaces in CI/CD 465

Getting ready • 465

How to do it… • 466

How it works… • 468

There's more… • 469

See also • 469

Building CI/CD for Terraform modules in Azure Pipelines 469

Getting ready • 470

How to do it… • 470

How it works… • 475

There's more… • 475

See also • 475

Building a workflow for publishing Terraform modules using GitHub Actions 476

Getting ready • 476

How to do it… • 476

How it works… • 478

There's more… • 481

See also • 481

Chapter 14: Using Terraform Cloud to Improve Team Collaboration 483

Technical requirements 485

Authenticating Terraform to Terraform Cloud • 486

Getting ready • 486

How to do it… • 487

There's more… • 489

See also • 490

Managing workspaces in Terraform Cloud 490

Getting ready • 491

How to do it… • 491

How it works… • 498

There's more… • 499

See also • 500

Using the remote backend in Terraform Cloud 500

Getting ready • 501

How to do it… • 502

How it works… • 505

There's more… • 506

See also • 506

Migrating Terraform State to Terraform Cloud 506

Getting ready • 507

How to do it… • 508

How it works… • 510

See also • 510

Using Terraform Cloud as a private module registry 510

Getting ready • 511

How to do it… • 513

How it works… • 515

There's more… • 515

See also • 515

Executing Terraform configuration remotely in Terraform Cloud 516

Getting ready • 516

How to do it… • 517

How it works… • 521

There's more… • 521

See also • 526

Checking the compliance of Terraform configurations using OPA in Terraform Cloud ... 526

Getting ready • 526

How to do it… • 528

There's more… • 535

See also • 535

Using integrated cost estimation for cloud resources 535

Getting ready • 536

How to do it... • 536

How it works... • 539

There's more... • 540

See also • 540

Integrating the Infracost run task during the Terraform Cloud run 540

Getting ready • 541

How to do it... • 543

How it works... • 547

There's more... • 547

See also • 547

Configuring Terraform Cloud with the Terraform TFE provider 547

Getting ready • 548

How to do it... • 549

There's more... • 551

See also • 552

Chapter 15: Troubleshooting Terraform Errors 553

Fixing interpolation errors 554

Getting ready • 554

How to do it... • 555

How it works... • 556

See also • 556

Fixing cycle errors 556

Getting ready • 556

How to do it... • 557

How it works... • 558

There's more... • 558

Fixing for_each errors 559

Getting ready • 559

How to do it... • 561

There's more... • 562

Fixing output errors 562

Getting ready • 562

How to do it... • 563

How it works... • 564

There's more... • 564

Appendix A: Terraform Cheat Sheet 565

Basic commands 565

Format Terraform configuration 566

Terraform providers management 566

Terraform dependency file 566

Basic workflow commands 567

Backend configuration 567

Validate configuration 568

Get outputs 568

Import resources 568

Terraform workspaces 568

Terraform debug 569

State management 569

Display Terraform graph dependencies 570

Taint/untaint resources 570

Terraform Cloud/Enterprise 570

Appendix B: Terraform Resources 571

Terraform official resources 571

Documentation • 571

Registry • 571

Providers development • 571

Terraform community resources 572

Terraform news feed 572

Terraform certifications and certification preparation 572

Terraform certification program pages • 572

Terraform certification preparation • 572

Other Books You May Enjoy 575

Index 579

Preface

Infrastructure as Code, more commonly known as IaC, is a practice that is a pillar of DevOps culture. IaC entails writing your desired architecture configuration in code. Among other advantages, IaC allows the automation of infrastructure deployments, which reduces or eliminates the need for manual intervention, and thus the risk of configuration errors, and the need to create templates and standardize infrastructure with modular and scalable code.

Among all the DevOps tools, there are many that allow IaC. One of them is Terraform, from HashiCorp, which is very popular today because, in addition to being open source and multi-platform, it has the following advantages:

- It allows you to preview the changes that will be applied to your infrastructure.
- It allows the parallelization of operations, considering the management of dependencies.
- It has a multitude of providers.

In this book dedicated to Terraform, we will first discuss the installation of Terraform, the writing of Terraform configurations, how to apply the Terraform workflow using the **command-line interface** (**CLI**), and how use Terraform modules.

Once configuration writing and commands in Terraform are understood, we will discuss Terraform's practical use for building infrastructure with the three leading cloud providers: Azure, AWS, and GCP. We will also explore how to use Terraform for Kubernetes in a chapter dedicated to this topic.

Finally, we will finish this book by looking at advanced uses of Terraform, including Terraform testing, integrating Terraform into a **continuous integration/continuous deployment** (**CI/CD**) pipeline, and using Terraform Cloud, which is Terraform's collaboration platform for teams and companies.

This book will guide you through several recipes on best practices for writing Terraform configurations and commands, and it will also cover recipes on Terraform's integration with other tools such as Terragrunt, kitchen-terraform, Tfsec, and Azure Pipelines.

Most of the Terraform configurations described in this book are based on the Azure provider, for illustration, but you can apply these recipes to all other Terraform providers.

In this second edition, the chapters have been completely redesigned, with over 50 new recipes and two brand new chapters: one on using Terraform with AWS and GCP, and another on Terraform and Kubernetes.

In writing this cookbook, I wanted to share my experience of real and practical Terraform-based scenarios that I have encountered while working with customers and companies over the years.

Who this book is for

This book is for developers, operators, and DevOps engineers looking to improve their workflow and use Infrastructure as Code. Experience with Microsoft Azure, Jenkins, shell scripting, and DevOps practices is required to get the most out of this Terraform book.

What this book covers

Chapter 1, *Setting Up the Terraform Environment*, details the different ways of installing Terraform manually, with scripts, or by using a Docker container, and it also details the Terraform migration configuration process.

Chapter 2, *Writing Terraform Configurations*, concerns the writing of Terraform configurations for a provider, variables, outputs, built-in functions, condition expressions, YAML file manipulation, and pre-and post-conditions.

Chapter 3, *Scaling Your Infrastructure with Terraform*, shows you how to build dynamic environments by going further with Terraform configuration writing using loops, maps, and collections.

Chapter 4, *Using Terraform with External Data*, explores how to use Terraform with external data and local files, and how to execute local programs and scripts with Terraform.

Chapter 5, *Managing Terraform State*, explains Terraform state management, including reading, moving, deleting, and importing resources into the Terraform state.

Chapter 6, *Applying a Basic Terraform Workflow*, explains the use of Terraform's CLI to validate the configuration, use outputs, destroy resources provisioned by Terraform, use workspaces, generate dependency graphs, and debug the execution of Terraform.

Chapter 7, *Sharing Terraform Configuration with Modules*, covers the creation, use, and sharing of Terraform modules, and shows testing module practices.

Chapter 8, *Provisioning Azure Infrastructure with Terraform*, illustrates the use of Terraform in a practical scenario with the cloud service provider Azure. It covers topics such as authentication, remote backends, ARM templates, Azure CLI execution, and Terraform configuration generation for an existing infrastructure.

Chapter 9, *Getting Starting to Provisioning AWS and GCP Infrastructure Using Terraform*, provides a starting point for provisioning AWS and GCP infrastructure using Terraform, and includes details on these providers, authentication, and remote backend storage.

Chapter 10, *Using Terraform for Docker and Kubernetes Deployment*, explains how to use Terraform to create Docker containers and deploy Kubernetes resources.

Chapter 11, *Running Test and Compliance Security on Terraform Configuration*, details Terraform configuration testing practices using several tools, including Tfsec, OPA, `terraform-compliance`, and Pester.

Chapter 12, *Deep-Diving into Terraform*, discusses topics that go further with Terraform, such as the execution of Terraform configuration tests, zero-downtime deployment, Terraform wrappers with Terragrunt, checking configuration using Git-Hook, and using the Terraform CDK as a developer.

Chapter 13, *Automating Terraform Execution in a CI/CD Pipeline*, explores local Terraform automation processes and implementing a CI/CD pipeline to apply Terraform configuration automatically.

Chapter 14, *Using Terraform Cloud to Improve Team Collaboration*, explains how to use Terraform Cloud to run Terraform in a team with the sharing of Terraform modules in a private registry, the use of remote backends for Terraform state, migrating Terraform state, running Terraform remotely, and integrating cost estimation.

Chapter 15, *Troubleshooting Terraform Errors*, lists several Terraform errors and explains how to resolve them.

The *Appendix A and B*, contains a Terraform CLI cheat sheet and Terraform resources list.

To get the most out of this book

The following is the list of software/hardware prerequisites for this book:

Software	OS requirements
Terraform Cli , version ≥1.5	Any OS
Terraform Cloud	NA (Browser)
Azure	Any Browser
Python version ≥ 3.11	Any OS
PowerShell scripting	Any OS
Shell scripting	Linux / WSL / MacOS
Golang Version ≥1.20	Any OS
Azure CLI	Any OS
Azure DevOps	Any Browser
GitHub	Any Browser
Git	Any OS
Ruby version ≥ 3.0.0	Any OS
Docker	Any OS
Terragrunt	Any OS
Jq	Any OS
Infracost	Any OS
kubectl / Helm	Any OS
Node.js	Any OS

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/P7a3G`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Execute the `terraform graph` command:”

A block of code is set as follows:

```
resource "azurerm_resource_group" "rg-app" {
  name     = "RG-APP-${terraform.workspace}"
  location = "westeurope"
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
terraform {
  backend "azurerm" {
    resource_group_name   = "RG-TFBACKEND"
    storage_account_name = "storagetfbackend"
    container_name       = "tfstate"
    key                  = "myapp.tfstate"
    access_key           = xxxxxx-xxxxx-xxx-xxxxx
  }
}
```

Any command-line input or output is written as follows:

```
terraform init
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Select **System info** from the **Administration** panel.”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share your thoughts

Once you've read *Terraform Cookbook, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781804616420`

2. Submit your proof of purchase
3. That’s it! We’ll send your free PDF and other benefits to your email directly

1

Setting Up the Terraform Environment

Before you start writing the Terraform configuration, it's necessary to understand the best practices to write configuration for **Infrastructure as Code** (**IaC**). Then, we can install and configure a local development environment. This development environment will allow us to write the Terraform configurations file and apply changes with Terraform.

In the recipes in this chapter, we will start to learn some of the most important IaC and Terraform best practices, then we will learn how to download and install Terraform manually on a Windows machine, as well as how to install it using a script on Windows and Linux. We will also learn how to use Terraform in a Docker container before learning how to upgrade Terraform providers.

In this chapter, we'll cover the following recipes:

- Overviewing Terraform best practices
- Downloading and installing Terraform on Windows manually
- Installing Terraform using Chocolatey on Windows
- Installing Terraform on Linux using the APT package manager
- Installing Terraform using a script on Linux
- Executing Terraform in a Docker container
- Writing Terraform configuration in Visual Studio Code
- Switching between multiple Terraform versions
- Upgrading Terraform providers

Let's get started!

Technical requirements

This chapter does not require that you have any specific technical knowledge. We will mainly use **graphical user interfaces** (**GUIs**) and simple Linux or Windows scripts executed in a terminal console. However, knowledge of Docker is recommended so that you can complete the *Executing Terraform in a Docker container* recipe.

Finally, for the **Integrated Development Environment** (**IDE**), which is the software we use to write the Terraform configuration, we will use Visual Studio Code, which is available for free at `https://code.visualstudio.com/`.

The source code for this chapter is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP01`.

Overviewing Terraform best practices

Before starting to learn how to install Terraform through the recipes presented in this chapter, it's necessary to understand the most common best practices of IaC with Terraform.

The first group of best practices is about IaC and are best practices for development in general:

- Store all Terraform configuration files and source code using a version control system such as GitHub, Azure DevOps, or Bitbucket.
- When the code is in Git, apply all good Git practices using branches, tags, commits, comments, and pull requests.
- Decouple your code file with multiple components; having a big monolithic code structure will make maintenance and deployment more difficult.
- Modularize and share common code for better reusability. For example, do not repeat the same code (as per the **Don't Repeat Yourself** (**DRY**) principle) and share the same business logic between components. We will learn about Terraform module implementation in *Chapter 7*, *Sharing Terraform Configuration with Modules*.
- Automate your infrastructure changes using the CI/CD pipeline that we explore in detail in *Chapter 13*, *Automating Terraform Execution in a CI/CD Pipeline*.

Then, we have best practices that are specific to Terraform:

- Write the required version of the Terraform binary explicitly in Terraform configuration and list all required providers with their required versions.

- Use remote backends to store and share the Terraform state file. We will learn how to do this in several recipes in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*, and in *Chapter 9*, *Getting Started with Provisioning AWS and GCP Infrastructure Using Terraform*.
- Don't use hard-coded values in configuration; use variables. We will learn more about this in *Chapter 2*, *Writing Terraform Configurations*.
- Add a description property in variables to communicate the role of the variable.
- Structure your Terraform project using these file-naming and organization conventions: `main.tf`, `version.tf`, `output.tf`, and `variables.tf`.
- Keep safe, sensitive data in code. We will learn more about this in *Chapter 2*, *Writing Terraform Configurations*.
- Protect the provider's authentication credentials. We will learn how to do this for Azure credentials in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*.
- Use naming conventions for all Terraform objects (resource, variable, output, and so on). For example, use an underscore to separate words in a resource's name.
- Have a Terraform well-formatted configuration using good indentation to make the code easy to read.
- Validate the code syntax with CLI commands and linters.
- Ensure Terraform providers are up to date to get all the fixes and new features.
- Test the configuration code with static code analysis and integration tests, which we will learn about in *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*.
- Always run the Terraform workflow with the `dry run` command to preview the changes that will be applied.

This is the main list of IaC and Terraform best practices and there are, of course, others that have not been mentioned. We will see them all in detail throughout the chapters and recipes in this book.

To learn more about Terraform best practices, I invite you to read these articles:

- Terraform recommended practices from *HashiCorp*: `https://www.terraform.io/cloud-docs/recommended-practices`
- Terraform best practice from *Google Cloud*: `https://cloud.google.com/docs/terraform/best-practices-for-terraform`
- Terraform best practices website: `https://www.terraform-best-practices.com/`
- Terraform best practices from *Brainboard*: `https://docs.brainboard.co/start/cloud-best-practices`

Now you are ready to install Terraform.

Downloading and installing Terraform on Windows manually

In this recipe, we will learn how to download and install Terraform on a local machine running a Windows operating system.

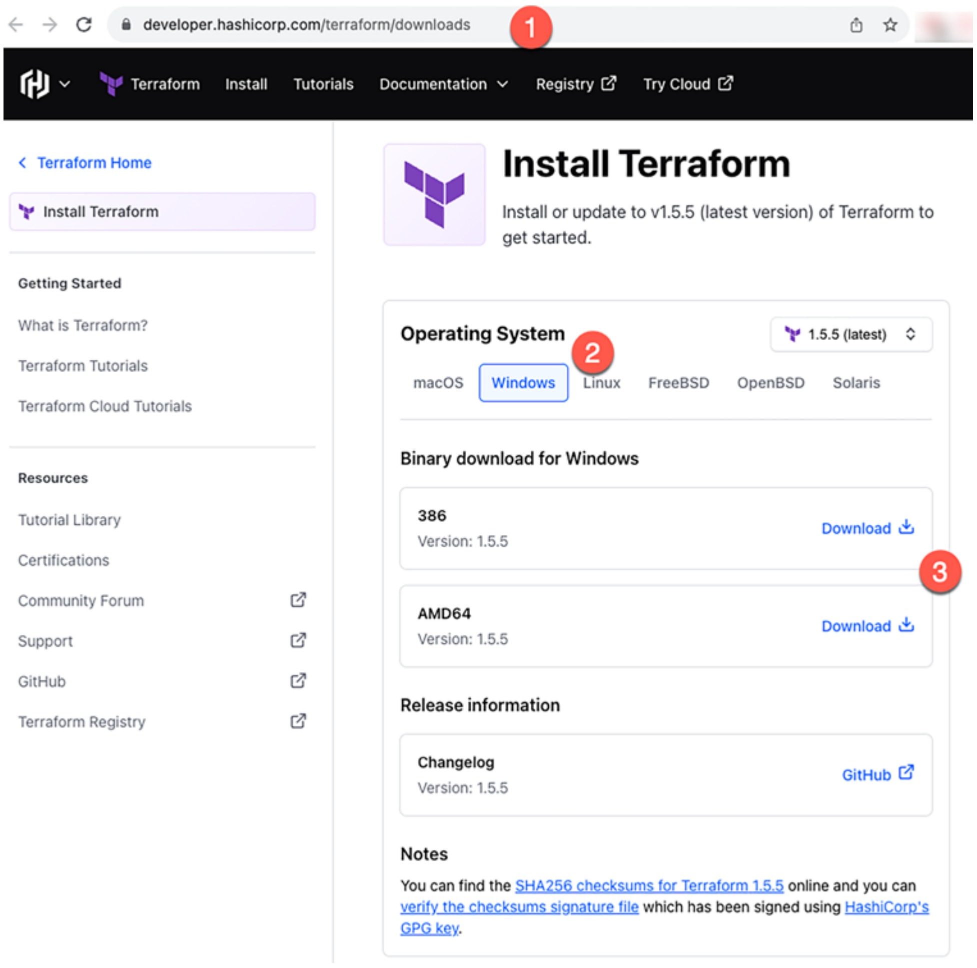
Getting ready

To complete this recipe, the only prerequisite is that you're on a Windows operating system.

How to do it…

Perform the following steps:

1. Open Windows File Explorer. Choose a location and create a folder called `Terraform`. We will use this to store the Terraform binary; for example, `C:\Terraform`.
2. Launch a web browser and go to `https://developer.hashicorp.com/terraform/downloads`.
3. Click on the **Windows** tab and then click on the **Amd64** link, which targets the Terraform ZIP package for the Windows 64-bit operating system. The ZIP package will be downloaded locally.



You can find the SHA256 checksums for Terraform 1.5.5 online and you can verify the checksums signature file which has been signed using HashiCorp's GPG key.

Figure 1.1: Download Windows Terraform binary

4. Unzip the content of the downloaded ZIP file into the `Terraform` folder that we created in *Step 1*:

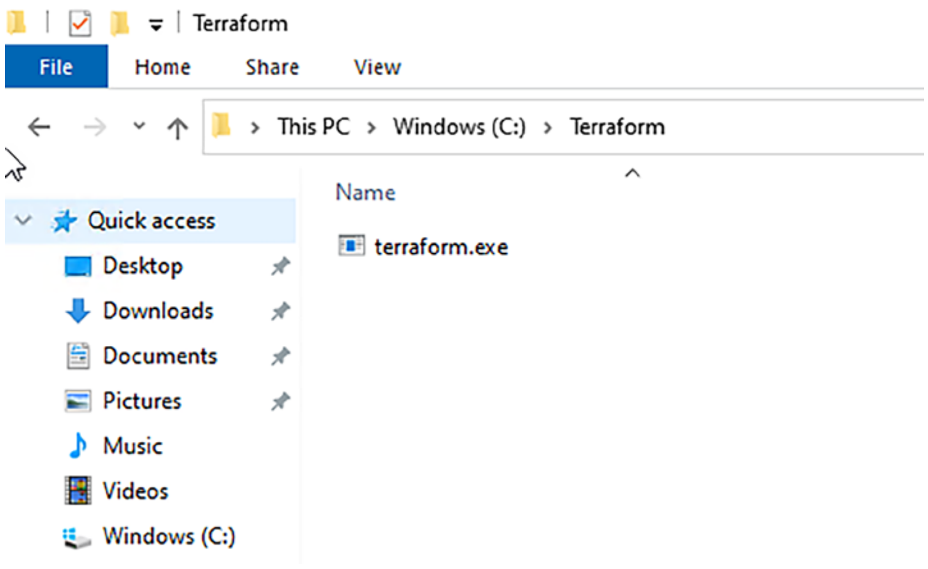


Figure 1.2: Copy Windows Terraform binary

The last thing we need to do to install Terraform is configure the `Path` environment variable by adding the path of the Terraform binary folder.

To complete this `Path` environment variable, follow these steps:

1. In File Explorer, right-click on the **This PC** menu and choose **Properties**:

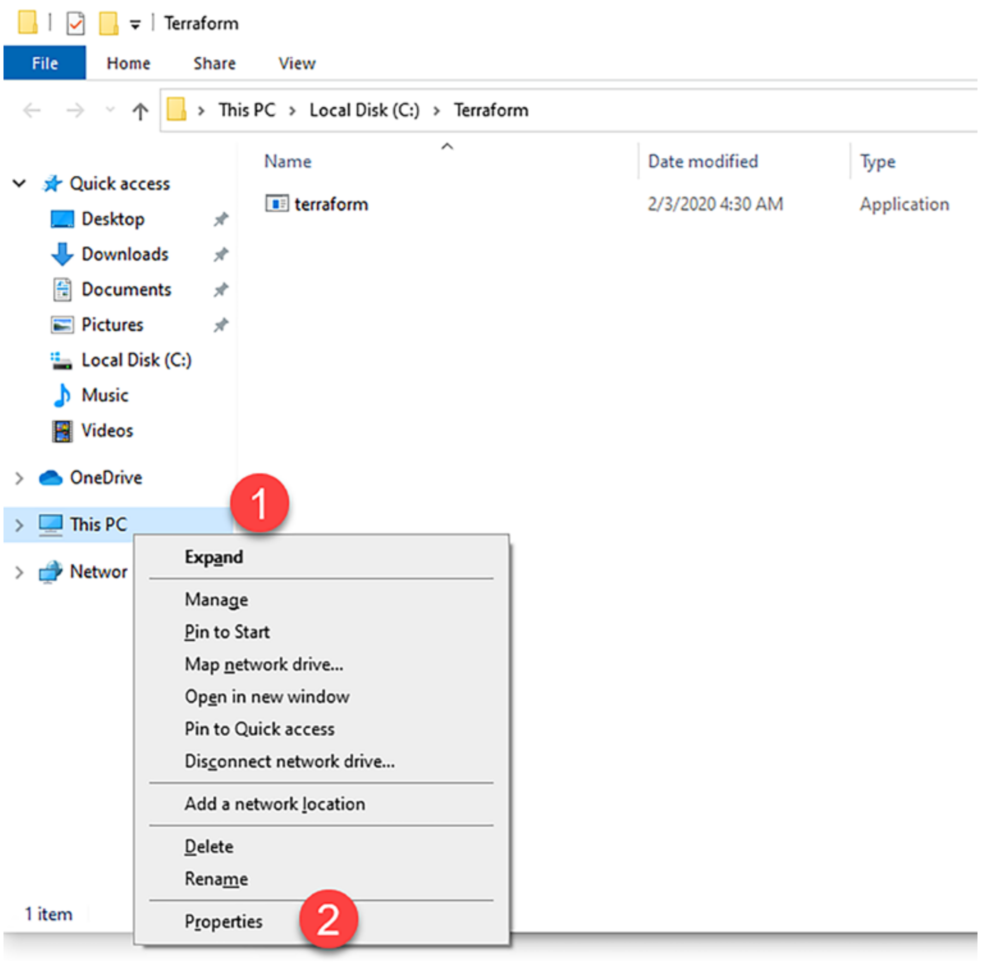
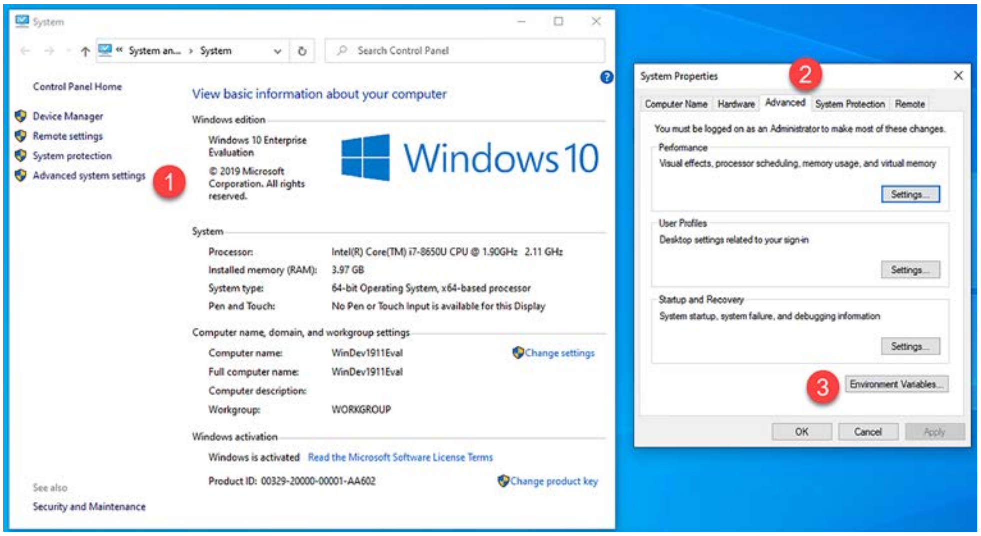


Figure 1.3: Open Windows properties

2. Click on the **Advanced system settings** link, then click on the **Advanced** tab and click the **Environment variables** button in the newly opened window:



View basic information about your computer

Windows edition

Windows 10 Enterprise Evaluation

© 2019 Microsoft Corporation. All rights reserved.

System

Processor:	Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz 2.11 GHz
Installed memory (RAM):	3.97 GB
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display

Computer name, domain, and workgroup settings

Computer name:	WinDev1911Eval
Full computer name:	WinDev1911Eval
Computer description:	
Workgroup:	WORKGROUP

Windows activation

Windows is activated Read the Microsoft Software License Terms

Product ID: 00329-20000-00001-AA602

System Properties

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles

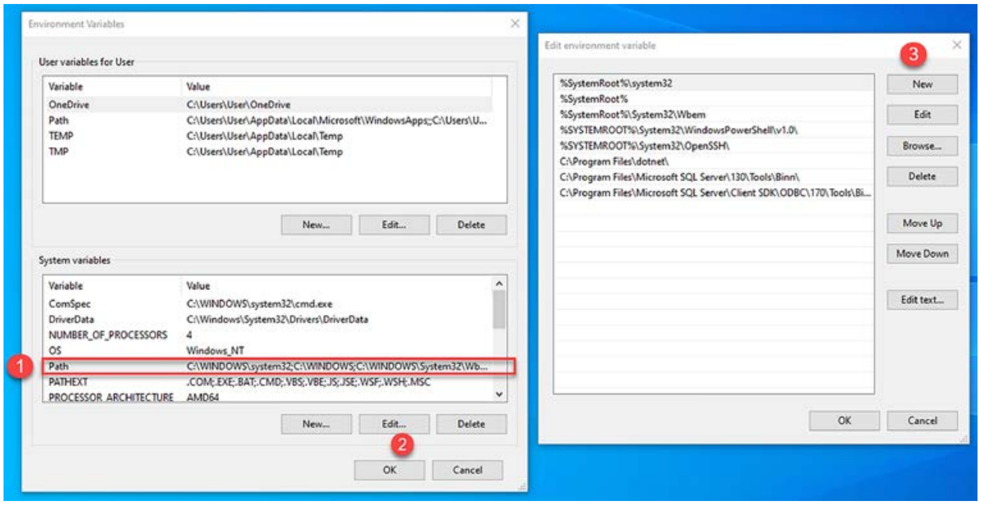
Desktop settings related to your sign-in

Startup and Recovery

System startup, system failure, and debugging information

Figure 1.4: Open environment variables options

3. When provided with a list of environments, select **User variables for User** or **System variables** (choose this option to apply the environment variable to all users of the workstation), and select the **Path** variable. Then, click on the **Edit** button and, in the newly opened window, click the **New** button:



User variables for User

Variable	Value
OneDrive	C:\Users\User\OneDrive
Path	C:\Users\User\AppData\Local\Microsoft\WindowsApps;;C:\Users\U...
TEMP	C:\Users\User\AppData\Local\Temp
TMP	C:\Users\User\AppData\Local\Temp

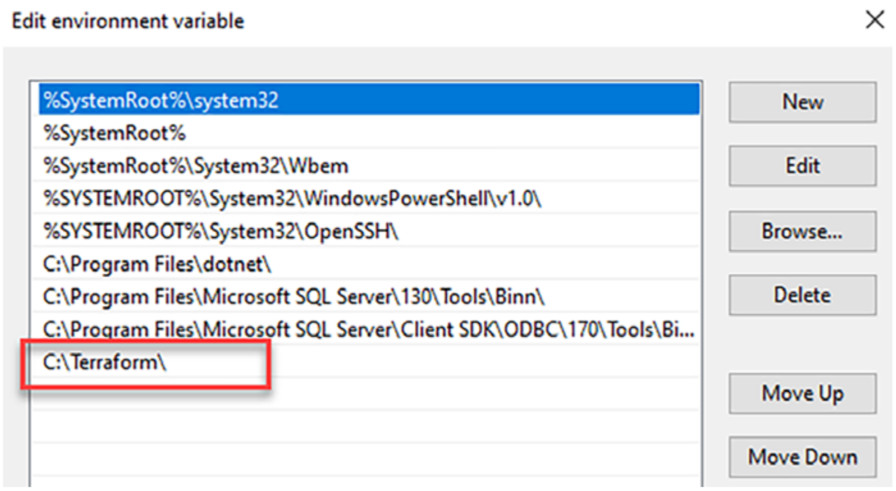
System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	4
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wb...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64

```
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files\dotnet\
C:\Program Files\Microsoft SQL Server\130\Tools\Binn\
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\170\Tools\Bi...
```

Figure 1.5: Select the Path environment variable

4. From the list of paths, add the folder we created, that is, `C:\Terraform\`:



Edit environment variable

```
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files\dotnet\
C:\Program Files\Microsoft SQL Server\130\Tools\Binn\
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\170\Tools\Bi...
C:\Terraform\
```

Figure 1.6: Complete the Path environment variable with Terraform CLI path

Finally, we validate all the open windows by clicking on the **OK** button, which is at the bottom of every open window.

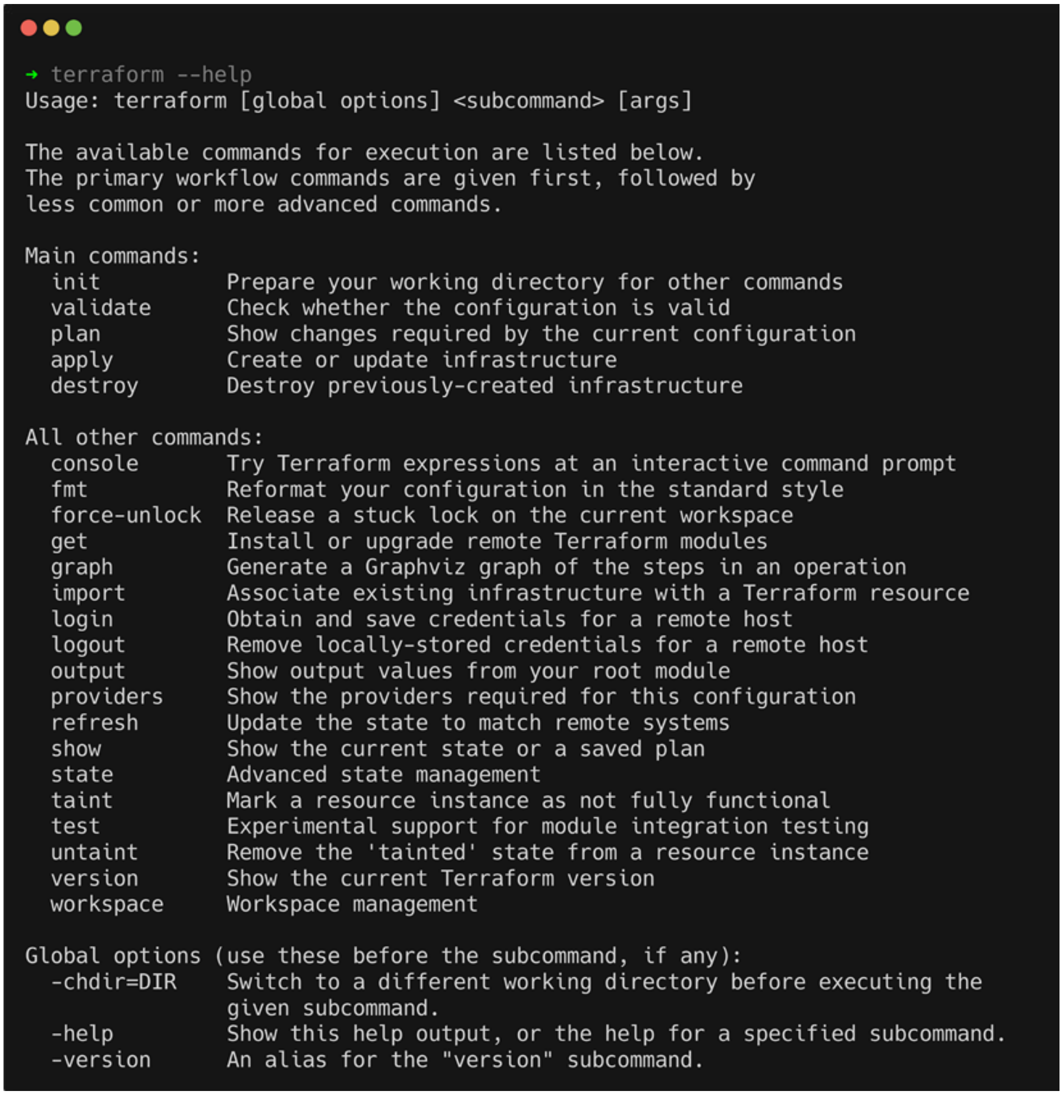
How it works...

Downloading and installing Terraform is simple and adding the path of the Terraform binary to the `PATH` environment variable makes it possible to execute the Terraform command line from any terminal location.

After completing all these steps, we can check that Terraform is working properly by opening a command-line terminal or PowerShell and executing the following command:

```
terraform --help
```

The result of executing the preceding command is shown in the following screenshot:



```
→ terraform --help
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure

All other commands:
  console       Try Terraform expressions at an interactive command prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  test          Experimental support for module integration testing
  untaint       Remove the 'tainted' state from a resource instance
  version       Show the current Terraform version
  workspace     Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing the
                given subcommand.
  -help         Show this help output, or the help for a specified subcommand.
  -version      An alias for the "version" subcommand.
```

Figure 1.7: Terraform commands list

This list of commands proves that Terraform has been installed correctly and that the Terraform command line can be accessed from any terminal location.

Installing Terraform using Chocolatey on Windows

In this recipe, we will learn how to install Terraform on a Windows machine using a script that uses the **Chocolatey** software package manager.

Getting ready

To complete this recipe, you'll need to be using a Windows operating system and have Chocolatey (`https://chocolatey.org/`) installed, which is a Windows software package manager.

If you don't have Chocolatey installed, you can easily install it by following these steps:

1. Open a PowerShell terminal in administrator mode, as shown in the following screenshot:

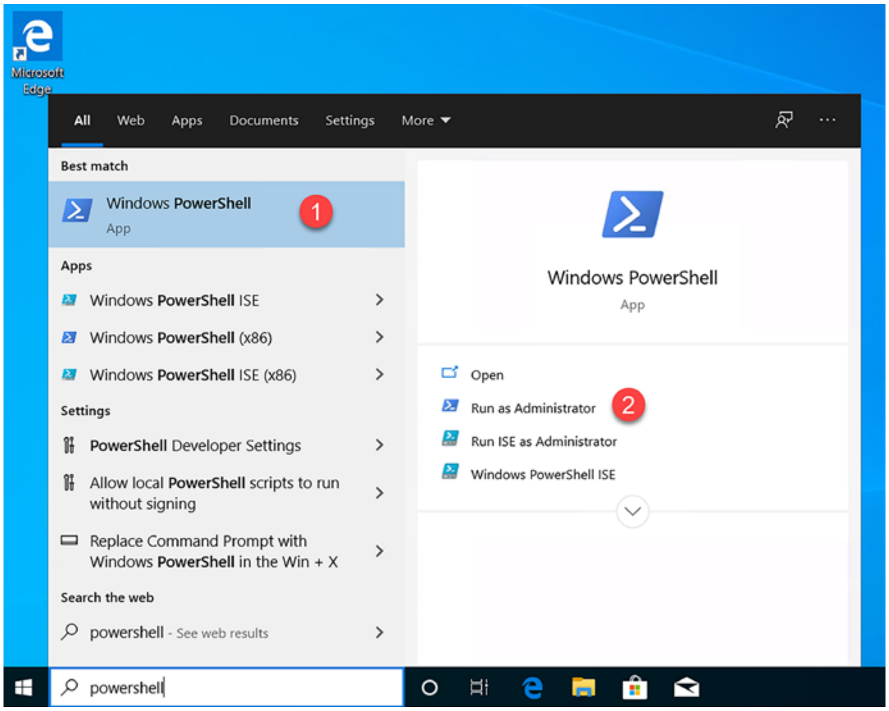


Figure 1.8: Running PowerShell as administrator

2. Then, execute the following script in the terminal:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.
Net.ServicePointManager]::SecurityProtocol = [System.Net.
ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/
install.ps1'))
```

The complete installation documentation for Chocolatey is available at `https://chocolatey.org/install`.

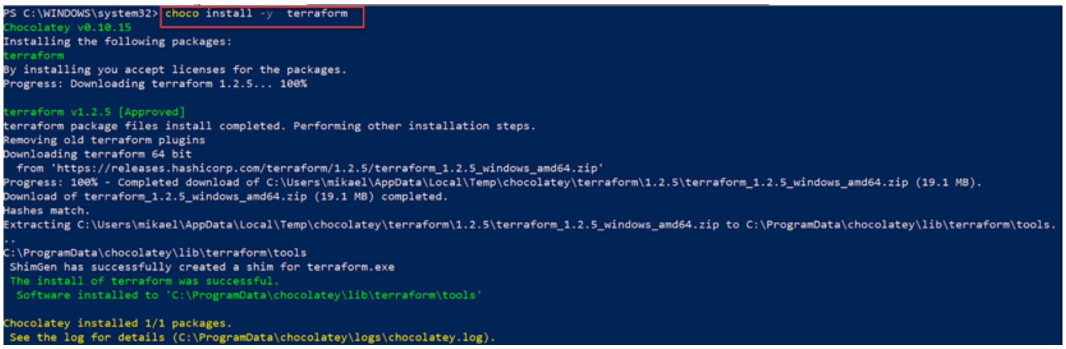
How to do it...

Perform the following steps:

1. Open a PowerShell command-line terminal in administrator mode.
2. Execute the following command:

```
choco install -y terraform
```

The following screenshot shows the execution of this command:



```
PS C:\WINDOWS\system32> choco install -y  terraform
Chocolatey v0.10.15
Installing the following packages:
terraform
By installing you accept licenses for the packages.
Progress: Downloading terraform 1.2.5... 100%

terraform v1.2.5 [Approved]
terraform package files install completed. Performing other installation steps.
Removing old terraform plugins
Downloading terraform 64 bit
  from 'https://releases.hashicorp.com/terraform/1.2.5/terraform_1.2.5_windows_amd64.zip'
Progress: 100% - Completed download of C:\Users\mikael\AppData\Local\Temp\chocolatey\terraform\1.2.5\terraform_1.2.5_windows_amd64.zip (19.1 MB).
Download of terraform_1.2.5_windows_amd64.zip (19.1 MB) completed.
Hashes match.
Extracting C:\Users\mikael\AppData\Local\Temp\chocolatey\terraform\1.2.5\terraform_1.2.5_windows_amd64.zip to C:\ProgramData\chocolatey\lib\terraform\tools.
..
C:\ProgramData\chocolatey\lib\terraform\tools
 ShimGen has successfully created a shim for terraform.exe
 The install of terraform was successful.
  Software installed to 'C:\ProgramData\chocolatey\lib\terraform\tools'

Chocolatey installed 1/1 packages.
 See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
```

Figure 1.9: Chocolatey install Terraform

The `-y` option is optional. It allows us to accept the license agreement automatically without user interaction.

How it works...

When Chocolatey installs the Terraform package, it executes the scripts in the package source code, available at `https://github.com/jamestoyer/chocolatey-packages/tree/master/terraform`.

Then, by executing the script available at `https://github.com/jamestoyer/chocolatey-packages/blob/master/terraform/tools/chocolateyInstall.ps1`, Chocolatey downloads the Terraform ZIP file into the binary directory of Chocolatey's packages, which is already included in the `PATH` environment variable.

There's more...

When upgrading Terraform, it is possible to upgrade it directly with Chocolatey by executing the `choco upgrade -y terraform` command.

By default, the `choco install` command installs the latest version of the mentioned package. It is also possible to specify the version by adding the `--version` option to the command, which in our case would give us the following:

```
choco install -y terraform --version "1.2.5"
```

In this example, we have specified that we want to install version 1.2.5 of Terraform and not the latest version.

See also

To learn about all the commands provided by Chocolatey, I suggest reading the following documentation: `https://chocolatey.org/docs/commands-reference#commands.`

Installing Terraform on Linux using the APT package manager

In the previous recipe, we learned how to install Terraform on Linux using a script. In this recipe, we will install Terraform on Linux using the **Advanced Package Tool** (**APT**) package manager, which is a common native package manager.

Note that in this recipe we will install Terraform on Ubuntu/Debian. To learn how to install Terraform on other Linux distributions, read the documentation here `https://www.hashicorp.com/official-packaging-guide`.

Getting ready

For this recipe, we need to have a Linux workstation and a terminal console.

How to do it...

To install Terraform on Linux using the package manager, execute the following script in the terminal console:

```
sudo apt update && sudo apt install gpg
wget -O- https://apt.releases.hashicorp.com/gpg | gpg --dearmor | \
sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg

gpg --no-default-keyring \
--keyring /usr/share/keyrings/hashicorp-archive-keyring.gpg \
--fingerprint

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update
sudo apt-get install terraform
```

The source code for this script is also available in this book's GitHub repository: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP01/install_terraform_package.sh`.

How it works...

In this script, the first command updates the packages index and installs the gpg tools. The script also downloads the GPG key and checks the fingerprint key.

Then the script registers the `HashiCorp` repository and updates the index of packages.

Finally, the script downloads and installs the Terraform package.

See also

- The documentation of the Terraform Official Packaging guide is available at `https://www.hashicorp.com/official-packaging-guide`
- The learning documentation of the installation of Terraform using package manager is available at `https://learn.hashicorp.com/tutorials/terraform/install-cli`

Installing Terraform using a script on Linux

In this recipe, we will learn how to install Terraform on a Linux machine using a script.

Getting ready

To complete this recipe, the only prerequisites are that you are running a Linux operating system and that you have an *unzip* utility installed. The `jq` utility must be also installed, and you can install it by using the following command:

```
apt update && apt install jq
```

`jq` is a tool used to perform queries on any JSON content, the documentation is available at `https://stedolan.github.io/jq/manual/`.

How to do it...

Perform the following steps:

1. Open a command-line terminal and execute the following script:

```
TERRAFORM_VERSION="1.3.2"
RES=$(curl -sS https://api.releases.hashicorp.com/v1/releases/
terraform/${TERRAFORM_VERSION})
BINARY_URL=$(echo $RES | jq -r '.builds[] | select (.os == "linux"
and .arch == "amd64") | .url')
wget $ BINARY_URL
unzip -o terraform_${TERRAFORM_VERSION}_linux_amd64.zip -d /usr/
local/bin
```

The source code for this script is also available in this book's GitHub repository: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP01/install_terraform_linux.sh`.

2. After executing this script, we can verify the installation of Terraform by executing the following command:

```
terraform --help
```

This command displays all the available commands of the Terraform CLI.

How it works...

In this script, in the first line, the `TERRAFORM_VERSION` variable is filled in with the Terraform version that we want to install. This variable is only mentioned here since we don't want to keep repeating the version of Terraform we're using throughout the script.

In this recipe, version 1.2.5 of Terraform is used, but we are free to modify this. But before changing the version, we recommend reading the Terraform changelog available at `https://github.com/hashicorp/terraform/blob/main/CHANGELOG.md` to see all changes.

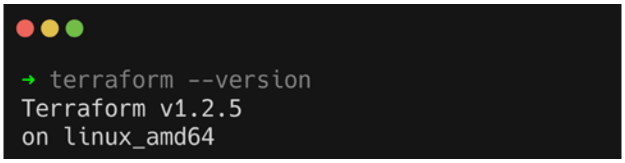
In line 2 of this script, we get the content of the *HashiCorp Releases* API for the release of the specified Terraform version (detailed here: `https://www.hashicorp.com/blog/announcing-the-hashicorp-releases-api`). Then, using `jq`, the script filters the returned JSON array to get only the Linux and amd64 release and returns the corresponding URL of the release.

Finally, the script downloads the package using `wget` and unzips the package into the `/usr/local/bin` folder, which is included by default in the `PATH` environment variable.

You can check that the version of Terraform you have installed corresponds to the one mentioned in the script by executing the following command:

```
terraform --version
```

This command displays all commands of Terraform, as shown in the following screenshot:



```
→ terraform --version
Terraform v1.2.5
on linux_amd64
```

Figure 1.10: Display the terraform help commands

As we can see here, Terraform is installed and operational.

There's more...

In this Terraform installation script, we have specified the version number of Terraform to be installed.

If you want to install the latest Terraform version without having to know the version number, it is also possible to dynamically retrieve the latest version number by changing the API in the above script by using the Releases API URI `https://api.releases.hashicorp.com/v1/releases/terraform/latest`. This API returns information about the latest version of Terraform.

This script studied in this recipe uses a basic workflow, we can use a more advanced script that downloads the Terraform binary, but also will check the integrity of the package.

This script is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP01/install_terraform_linux_v2.sh`, and for more details about checking the package, read the documentation at `https://learn.hashicorp.com/tutorials/terraform/verify-archive?in=terraform/cli`.

In this recipe, we learned how to install Terraform on Linux using a script. In the next recipe, we will learn how to install Terraform using a Linux package manager.

See also

For more information on verifying the downloaded package, you can consult the HashiCorp documentation at `https://www.hashicorp.com/security.html`.

Executing Terraform in a Docker container

In the previous recipes of this chapter, we discussed how to install Terraform locally, either manually or via a script, for Windows and Linux.

In this recipe, we will learn how to run Terraform in a Docker container, which will allow us to enjoy the following benefits:

- There is no need to install Terraform locally
- We can have a Terraform runtime environment that's independent of the local operating system
- We can test our Terraform configuration with different versions of Terraform

Let's get started!

Getting ready

To complete this recipe, you'll need to know about Docker and its commands, as well as how to write **Dockerfiles**. To learn basic expressions of **Dockerfiles**, please read the documentation: `https://docs.docker.com/get-started/overview/`.

On our local computer, we installed Docker using a tool called Docker Desktop for Windows.

For Docker installation guides for other operating systems, please read the Docker installation documentation at `https://docs.docker.com/get-docker/`.

We have already written a Terraform configuration file, which will not be detailed here. The source code is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP01/terraform-docker/main.tf`. This will be executed in our Docker container.

You will also need the `init`, `plan`, and `apply` Terraform commands, which will not be explained in the context of this recipe.

How to do it...

Perform the following steps:

1. At the root of the folder that contains the Terraform configuration, we need to create a Dockerfile that contains the following code:

```
FROM golang:latest
ENV TERRAFORM_VERSION=1.2.5
RUN apt-get update && apt-get install unzip \
    && curl -Os https://releases.hashicorp.com/
terraform/${TERRAFORM_VERSION}/terraform_${TERRAFORM_VERSION}_linux_
amd64.zip \
    && curl -Os https://releases.hashicorp.com/
terraform/${TERRAFORM_VERSION}/terraform_${TERRAFORM_VERSION}_
SHA256SUMS \
    && curl https://keybase.io/hashicorp/pgp_keys.asc | gpg
--import \    && curl -Os https://releases.hashicorp.com/
terraform/${TERRAFORM_VERSION}/terraform_${TERRAFORM_VERSION}_
SHA256SUMS.sig \
    && gpg --verify terraform_${TERRAFORM_VERSION}_SHA256SUMS.sig
terraform_${TERRAFORM_VERSION}_SHA256SUMS \
    && shasum -a 256 -c terraform_${TERRAFORM_VERSION}_SHA256SUMS
2>&1 | grep "${TERRAFORM_VERSION}_linux_amd64.zip:\sOK" \
    && unzip -o terraform_${TERRAFORM_VERSION}_linux_amd64.zip -d /
usr/bin
RUN mkdir /tfcode
COPY . /tfcode
WORKDIR /tfcode
```

For this script, we use the Linux script we learned in the *Installing Terraform on Linux using a script* recipe to install Terraform. We can also use another method to install Terraform using a package manager, as we learned in the *Installing Terraform using the package manager* recipe.

This source code is also available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP01/terraform-docker/Dockerfile`.

2. Next, we need to create a new Docker image by executing the `docker build` command in a terminal:

```
docker build -t terraform-code:v1.0
```

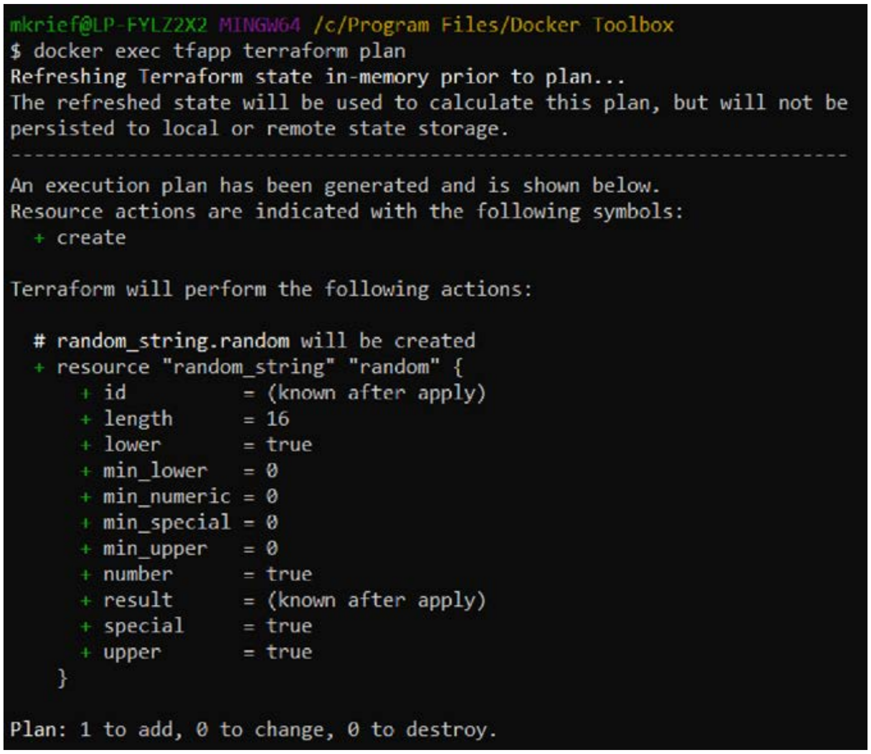
3. Then, we need to instantiate a new container of this image. To do this, we will execute the `docker run` command:

```
docker run -it -d --name tfapp terraform-code:v1.0 /bin/bash
```

4. Now, we can execute the Terraform commands in our container by using the following commands:

```
docker exec tfapp terraform init
docker exec tfapp terraform plan
docker exec tfapp terraform apply --auto-approve
```

The following screenshot shows a part of the output of executing these commands (`terraform plan`):



```
mkrief@LP-FYLZ2X2 MINGW64 /c/Program Files/Docker Toolbox
$ docker exec tfapp terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # random_string.random will be created
  + resource "random_string" "random" {
      + id          = (known after apply)
      + length      = 16
      + lower       = true
      + min_lower   = 0
      + min_numeric = 0
      + min_special = 0
      + min_upper   = 0
      + number      = true
      + result      = (known after apply)
      + special     = true
      + upper       = true
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Figure 1.11: Run Terraform in Docker container

In this command execution, we can see the `terraform plan` command that has been executed, with a preview of the changes.

How it works…

In *Step 1*, we write the composition of the Docker image in the Dockerfile. We do this as follows:

1. We use a Golang base image.
2. We initialize the `TERRAFORM_VERSION` variable with the version of Terraform to be installed.
3. We write the same Terraform installation script we wrote in the *Installing Terraform using a script on Linux* recipe.
4. We copy the Terraform configuration from our local file into a new folder located in the image.
5. We specify that our workspace will be our new folder.

Then, in *Steps 2* and *3*, we create a Docker `terraform-code` image with a `v1.0` tag. This tag is used to version our Terraform configuration. Then, we create a `tfapp` instance of this image, which runs with the Bash tool.

Finally, in *Step 4*, in the `tfapp` instance, we execute the Terraform commands in our container workspace.

There's more…

In this recipe, we studied how to write, build, and use a Docker image that contains the Terraform binary. With this, it is possible to complete this image with other tools, such as **Terragrunt**, that are also used to develop the Terraform configuration file.

If you just want to use Terraform, you can use the official image provided by HashiCorp. This is public and available on Docker Hub at `https://hub.docker.com/r/hashicorp/terraform/`. If you need more tools, you can use this official image as your base FROM image in your Dockerfile.

See also

- The full Docker commands documentation at `https://docs.docker.com/engine/reference/run/`
- For an introduction to Docker, please refer to the book *Learning DevOps Second Edition*, which is available at `https://www.packtpub.com/product/learning-devops/9781801818964`

Switching between multiple Terraform versions

In the previous recipes, we learned how to install one specific version of Terraform manually or by script.

In some scenarios, we need to have different Terraform versions installed on the same workstation:

- When we work on multiple Terraform configuration projects, it's often required to have different versions of Terraform binary used by each of these projects
- When we want to test new versions of the Terraform binary

All previously described methods of installation only allow us to use one Terraform version at a time.

To solve this problem, we have different solutions:

- To rename the Terraform binary by including the Terraform version at the end of the binary. For example, for Terraform version 1.0.5, download the corresponding binary and extract it as shown in the *Downloading and installing Terraform on Windows manually* recipe. Rename the target file `terraform1.0.5.exe`. Then run all commands as `terraform1.0.5.exe <command>`.
- The second solution is to execute Terraform in a Docker container, as we learned in the *Executing Terraform in a Docker container* recipe.
- Another solution that will learn in detail in this recipe is to use a tool named `tfenv`.

`tfenv` is an open-source project that acts as a Terraform version manager. It allows us to list, install, and use multiple versions of the Terraform binary on the same workstation.

In this recipe, we will learn how to use `tfenv` for the following operations:

- The installation of `tfenv`
- The download of multiple versions of Terraform with `tfenv`
- Finally, the configuration of Terraform to use one of the downloaded versions of Terraform

So, by following this recipe, we will learn how is it possible for one workstation to work on different projects using different versions of Terraform.

Let's get started!

Getting ready

As mentioned in the `tfenv` documentation at `https://github.com/tfutils/tfenv`, it's compatible only with macOS, Linux, and WSL for Windows users. In this recipe, we will use `tfenv` on WSL, which is the same procedure as Linux.

To complete this recipe, you need to be on a Linux OS (like WSL) and to have Git installed.

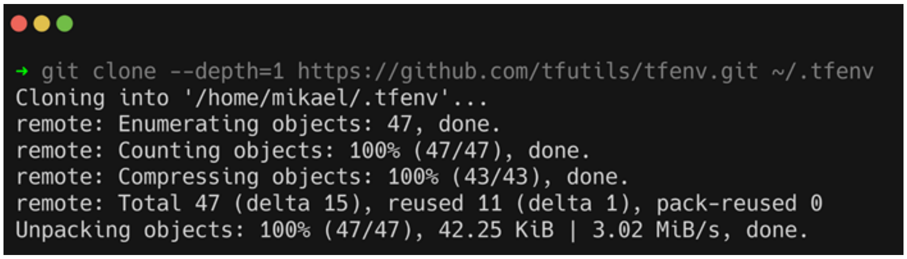
How to do it…

To install `tfenv`, perform the following steps in your Linux terminal:

1. Run the following command:

   ```
   git clone --depth=1 https://github.com/tfutils/tfenv.git ~/.tfenv
   ```

   The following screenshot shows this `git clone` execution:

   ```
   ➜ git clone --depth=1 https://github.com/tfutils/tfenv.git ~/.tfenv
   Cloning into '/home/mikael/.tfenv'...
   remote: Enumerating objects: 47, done.
   remote: Counting objects: 100% (47/47), done.
   remote: Compressing objects: 100% (43/43), done.
   remote: Total 47 (delta 15), reused 11 (delta 1), pack-reused 0
   Unpacking objects: 100% (47/47), 42.25 KiB | 3.02 MiB/s, done.
   ```

   Figure 1.12: Install Tfenv with git clone

2. Then run this command:

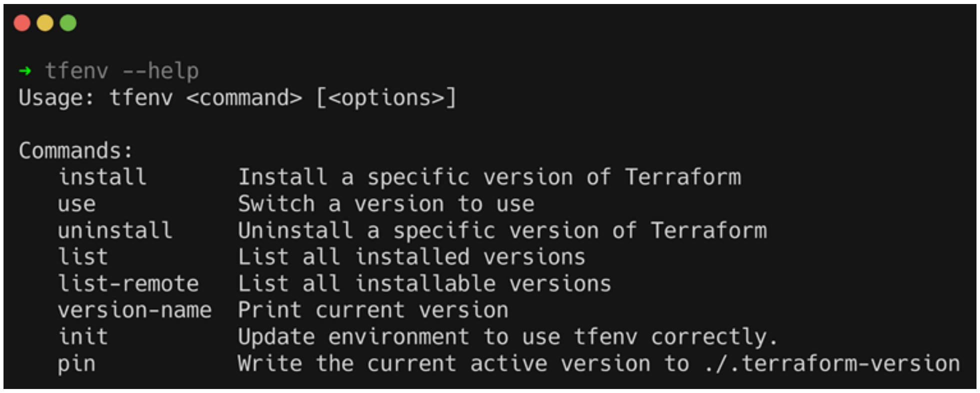
   ```
   echo 'export PATH="$HOME/.tfenv/bin:$PATH"' >> ~/.bashrc
   ```

   `tfenv` is now installed and can be run from any folder in the workstation.

3. To check that the installation is running, use this command:

   ```
   tfenv --help
   ```

The following screenshot shows the execution of the `tfenv -help` command:



```
→ tfenv --help
Usage: tfenv <command> [<options>]

Commands:
   install       Install a specific version of Terraform
   use           Switch a version to use
   uninstall     Uninstall a specific version of Terraform
   list          List all installed versions
   list-remote   List all installable versions
   version-name  Print current version
   init          Update environment to use tfenv correctly.
   pin           Write the current active version to ./.terraform-version
```

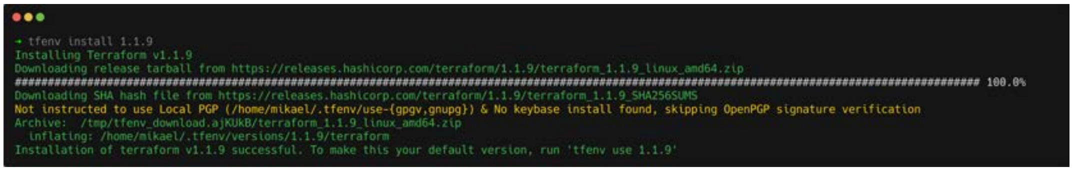
Figure 1.13: tfenv help command

This command displays the list of the available `tfenv` commands.

Then, in the second step, we will install multiple versions of Terraform with `tfenv` using the `tfenv install <version>` command. For example, to install version 1.1.9 of Terraform, we run the following:

```
tfenv install 1.1.9
```

The following screenshot shows the execution of the `tfenv install` command:



```
→ tfenv install 1.1.9
Installing Terraform v1.1.9
Downloading release tarball from https://releases.hashicorp.com/terraform/1.1.9/terraform_1.1.9_linux_amd64.zip
######################################################################################################################################## 100.0%
Downloading SHA hash file from https://releases.hashicorp.com/terraform/1.1.9/terraform_1.1.9_SHA256SUMS
Not instructed to use Local PGP (/home/mikael/.tfenv/use-{gpgv,gnupg}) & No keybase install found, skipping OpenPGP signature verification
Archive:  /tmp/tfenv_download.ajKUkB/terraform_1.1.9_linux_amd64.zip
  inflating: /home/mikael/.tfenv/versions/1.1.9/terraform
Installation of terraform v1.1.9 successful. To make this your default version, run 'tfenv use 1.1.9'
```

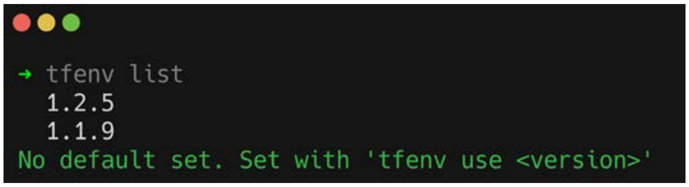
Figure 1.14: tfenv install Terraform version

For the lab in this recipe, we will install another Terraform version, 1.2.5. To do this, we run the `tfenv install 1.2.5` command.

To check the installed version, run the following command:

```
tfenv list
```

The following screenshot shows the execution of the `tfenv list` command:



```
→ tfenv list
  1.2.5
  1.1.9
No default set. Set with 'tfenv use <version>'
```

Figure 1.15: tfenv list Terraform version

We can see the two installed versions are 1.2.5 and 1.1.9.

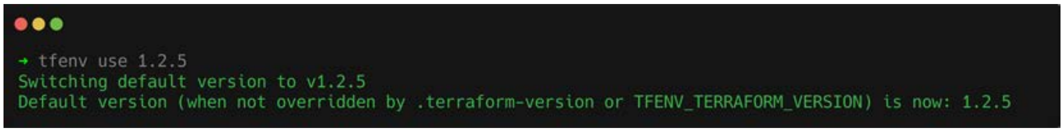
Now, in the last step, we will learn how to use one of the specific Terraform versions installed by `tfenv`.

To do this, we have two possibilities: choose one Terraform version that will be used all Terraform configurations in the same workstation, or choose a different Terraform version for each Terraform configuration in the same workstation.

To select one version of Terraform for all Terraform configurations on your workstation, run the following command:

```
tfenv use 1.2.5
```

The following screenshot shows the execution of the `tfenv use` command:



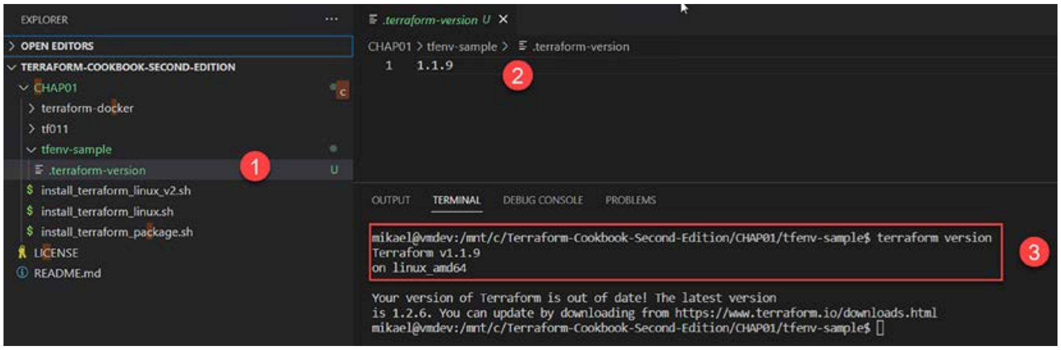
```
→ tfenv use 1.2.5
Switching default version to v1.2.5
Default version (when not overridden by .terraform-version or TFENV_TERRAFORM_VERSION) is now: 1.2.5
```

Figure 1.16: tfenv use Terraform version

To select one specific Terraform version for one Terraform configuration, follow these steps:

1. In the root of the Terraform configuration, create a new file called `.terraform-version`.
2. In this file, write the version of Terraform to use for this Terraform configuration, for example, 1.1.9.
3. In this folder, in the console, run the command to display the version of Terraform for this configuration:

   ```
   terraform version
   ```


EXPLORER
OPEN EDITORS
TERRAFORM-COOKBOOK-SECOND-EDITION
CHAP01
terraform-docker
tf011
tfenv-sample
.terraform-version
install_terraform_linux_v2.sh
install_terraform_linux.sh
install_terraform_package.sh
LICENSE
README.md
.terraform-version
CHAP01 > tfenv-sample > .terraform-version
1
2
3
OUTPUT
TERMINAL
DEBUG CONSOLE
PROBLEMS

```
1    1.1.9
```

```
mikael@vmdev:/mnt/c/Terraform-Cookbook-Second-Edition/CHAP01/tfenv-sample$ terraform version
Terraform v1.1.9
on linux_amd64

Your version of Terraform is out of date! The latest version
is 1.2.6. You can update by downloading from https://www.terraform.io/downloads.html
mikael@vmdev:/mnt/c/Terraform-Cookbook-Second-Edition/CHAP01/tfenv-sample$
```

Figure 1.17: tfenv Terraform version

We can see in the output that in this folder, the version of Terraform that will be used is 1.1.9. Although the version used here is 1.1.9, we could also still use version 1.2.5 for a different project.

How it works...

In the first part of the recipe, we installed `tfenv` from the GitHub repository and ran the `git clone` command to import the source code locally inside the `.tfenv` folder.

To complete the installation, using the `export` command, we added the **PATH** environment variables of the folder path of the `tfenv` binary (also added in the `.bashrc` file to be available in any session).

Then, in the second part, we used `tfenv` to install two different versions of the Terraform binary using the `tfenv install` command, and we displayed the list of two installed versions using the `tfenv list` command.

Finally, we learned how to use installed Terraform versions either by using the command line with the `tfenv use` command, or by using a `.terraform-version` file that we put in a folder containing a Terraform configuration.

There's more...

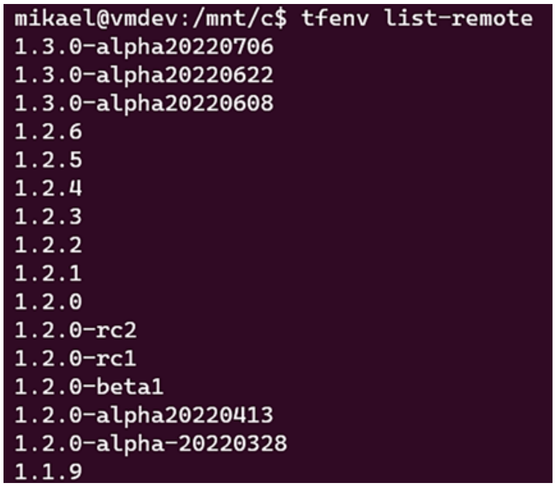
To specify a version of Terraform to use, we can also use a variable environment called `TFENV_TERRAFORM_VERSION` that allows the user to provide Terraform dynamically, for example, in a CI/CD pipeline.

Be careful: using different versions of the Terraform binary on the same Terraform configuration can have an impact on the compatibility of the Terraform state file. However, it is worth noting that all recent versions of Terraform (perhaps even back to 0.12 or 0.13, but certainly the v1 series) have built-in safety mechanisms that will prevent you from using an older version of Terraform if the state was modified using a later version, hence preventing conflicts between versions, so this should no longer be a worry for end-users.

There are other useful `tfenv` commands.

To list all available Terraform versions that can be installed with `tfenv`, we can use the `tfenv list-remote` command.

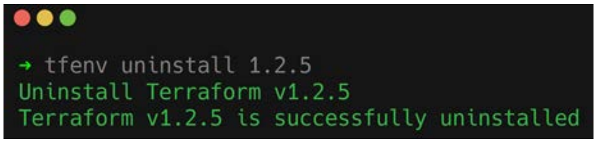
Here is an extract of the output of this command:



```
mikael@vmdev:/mnt/c$ tfenv list-remote
1.3.0-alpha20220706
1.3.0-alpha20220622
1.3.0-alpha20220608
1.2.6
1.2.5
1.2.4
1.2.3
1.2.2
1.2.1
1.2.0
1.2.0-rc2
1.2.0-rc1
1.2.0-beta1
1.2.0-alpha20220413
1.2.0-alpha-20220328
1.1.9
```

Figure 1.18: tfenv list all Terraform versions

To uninstall an installed version of the Terraform binary with `tfenv`, we can use the `tfenv uninstall <version>` command:



```
➜ tfenv uninstall 1.2.5
Uninstall Terraform v1.2.5
Terraform v1.2.5 is successfully uninstalled
```

Figure 1.19: tfenv uninstall

Now the `tfenv list` command returns only version 1.1.9.

See also

- The official GitHub repository and complete documentation of `tfenv`: `https://github.com/tfutils/tfenv`
- To use `tfenv` in a Docker container, here is a list of Docker images with `tfenv` pre-installed: `https://github.com/DockerToolbox/tfenv`

Upgrading Terraform providers

One of the best practices of Terraform is to ensure the Terraform configuration is up to date with the most recent version of Terraform providers used in the configuration.

If you encounter a problem with the Terraform `init` command, try updating your Terraform provider. We will show you how to do this in this section.

For information, `init` is the first step in the Terraform workflow. For more details on `init`, see the documentation here: `https://www.terraform.io/cli/commands/init`.

In this recipe, we will see how to update a provider in your Terraform configuration.

Let's get started!

Getting ready

To complete this recipe, you'll need to have Terraform installed, and we start with the existing following basic Terraform configuration written in `main.tf`:

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 2.67.0"
    }
  }
}
```

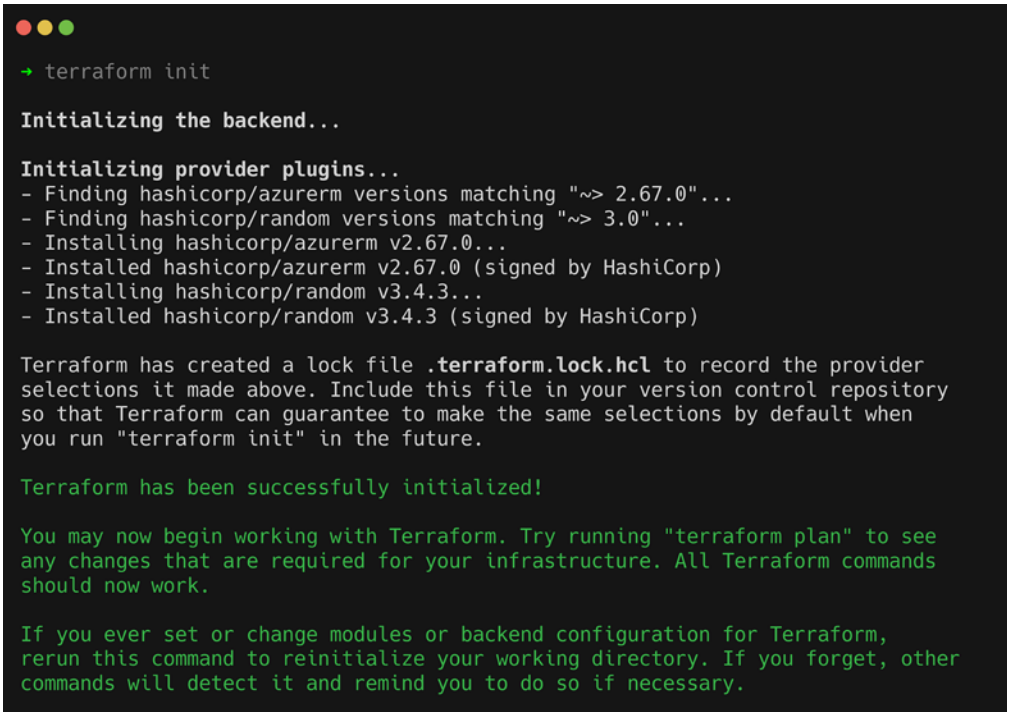
In this configuration, we explicitly mention that we use the `azurerm` provider with version 2.67.0.

How to do it...

As the first step, inside the folder that contains the `main.tf` file, run the `init` command as follows:

```
terraform init
```

The following screenshot shows the output of this command:

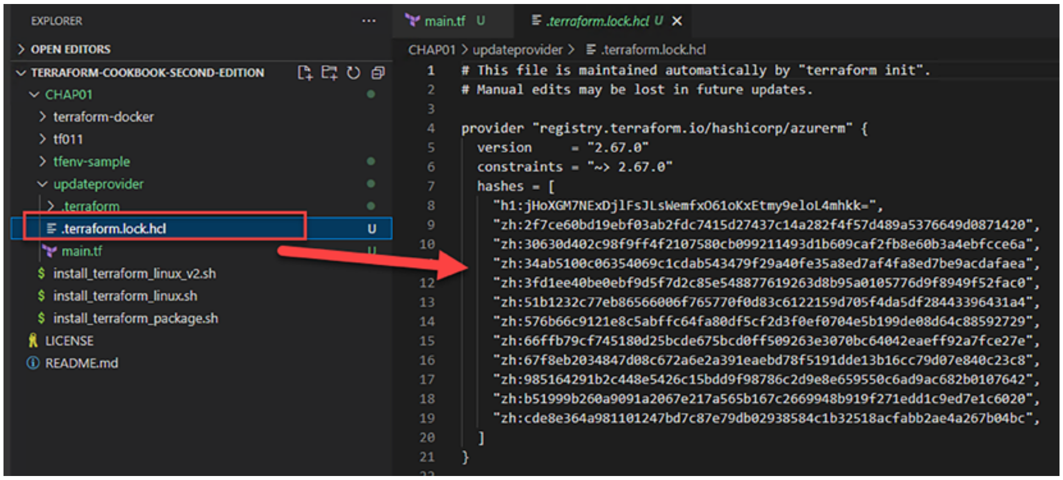


```
➜ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "~> 2.67.0"...
- Finding hashicorp/random versions matching "~> 3.0"...
- Installing hashicorp/azurerm v2.67.0...
- Installed hashicorp/azurerm v2.67.0 (signed by HashiCorp)
- Installing hashicorp/random v3.4.3...
- Installed hashicorp/random v3.4.3 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Figure 1.20: terraform init command

So far, everything is OK: we get the expected result. At the end of the execution of this command, we can see that Terraform has created a new file named `.terraform.lock.hcl` that contains hashes and versions of the Terraform providers.

The following screenshot shows this file, `.terraform.lock.hcl`:


EXPLORER
OPEN EDITORS
TERRAFORM-COOKBOOK-SECOND-EDITION
CHAP01
terraform-docker
tf011
tfenv-sample
updateprovider
.terraform
.terraform.lock.hcl
main.tf
install_terraform_linux_v2.sh
install_terraform_linux.sh
install_terraform_package.sh
LICENSE
README.md
main.tf U
.terraform.lock.hcl U
CHAP01 > updateprovider > .terraform.lock.hcl

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/azurerm" {
  version     = "2.67.0"
  constraints = "~> 2.67.0"
  hashes = [
    "h1:jHoXGM7NExDjlFsJLsWemfxO61oKxEtmy9eloL4mhkk=",
    "zh:2f7ce60bd19ebf03ab2fdc7415d27437c14a282f4f57d489a5376649d0871420",
    "zh:30630d402c98f9ff4f2107580cb099211493d1b609caf2fb8e60b3a4ebfcce6a",
    "zh:34ab5100c06354069c1cdab543479f29a40fe35a8ed7af4fa8ed7be9acdafaea",
    "zh:3fd1ee40be0ebf9d5f7d2c85e548877619263d8b95a0105776d9f8949f52fac0",
    "zh:51b1232c77eb86566006f765770f0d83c6122159d705f4da5df28443396431a4",
    "zh:576b66c9121e8c5abffc64fa80df5cf2d3f0ef0704e5b199de08d64c88592729",
    "zh:66ffb79cf745180d25bcde675bcd0ff509263e3070bc64042eaeff92a7fce27e",
    "zh:67f8eb2034847d08c672a6e2a391eaebd78f5191dde13b16cc79d07e840c23c8",
    "zh:985164291b2c448e5426c15bdd9f98786c2d9e8e659550c6ad9ac682b0107642",
    "zh:b51999b260a9091a2067e217a565b167c2669948b919f271edd1c9ed7e1c6020",
    "zh:cde8e364a981101247bd7c87e79db02938584c1b32518acfabb2ae4a267b04bc",
  ]
}
```

Figure 1.21: terraform.lock.hcl file

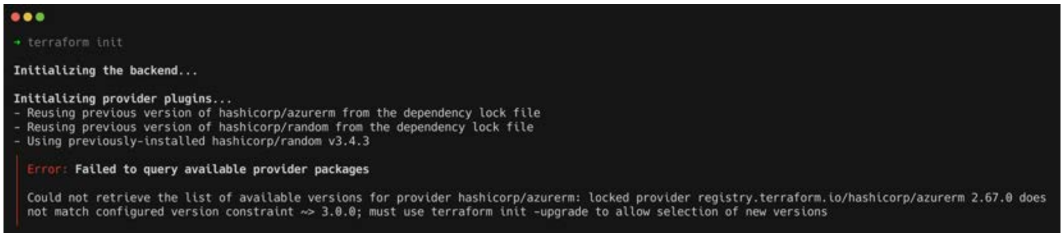
This file contains Terraform provider version information.

The `terraform init` command also downloads the binary of the provider inside the `.terraform` folder.

The second step is to upgrade the `azurerm` provider by updating the version number to 3.0.0 with the following configuration:

```
terraform {
  required_version = ">= 1.0.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0.0"
    }
  }
}
```

Then, if we re-run the simple `init` command, `terraform init`, the output of the command returns the following error:



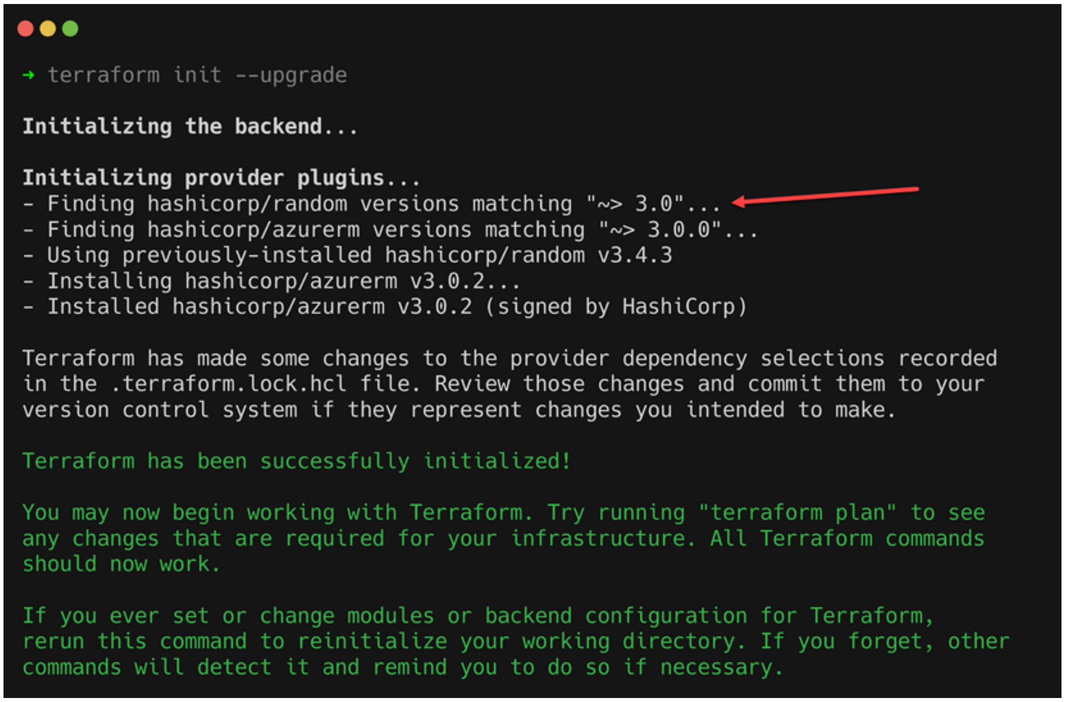
```
→ terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Reusing previous version of hashicorp/random from the dependency lock file
- Using previously-installed hashicorp/random v3.4.3
│
│ Error: Failed to query available provider packages
│
│ Could not retrieve the list of available versions for provider hashicorp/azurerm: locked provider registry.terraform.io/hashicorp/azurerm 2.67.0 does
│ not match configured version constraint ~> 3.0.0; must use terraform init -upgrade to allow selection of new versions
```

Figure 1.22: terraform init command with a provider error

Finally, to fix this error and make the configuration work properly, run the following `terraform init` command:

```
terraform init -upgrade
```

The following screenshot shows the execution of the `terraform init -upgrade` command:



```
→ terraform init --upgrade

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/random versions matching "~> 3.0"...
- Finding hashicorp/azurerm versions matching "~> 3.0.0"...
- Using previously-installed hashicorp/random v3.4.3
- Installing hashicorp/azurerm v3.0.2...
- Installed hashicorp/azurerm v3.0.2 (signed by HashiCorp)

Terraform has made some changes to the provider dependency selections recorded
in the .terraform.lock.hcl file. Review those changes and commit them to your
version control system if they represent changes you intended to make.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Figure 1.23: terraform init with upgrade option

Our configuration is now updated with the most recent provider.

It is recommended to maintain the lock file in your VCS (such as Git) to ensure that the same configuration can be consistently applied using the exact same provider version and also to add an extra security layer, preventing possible **Man in the Middle** (**MITM**) attacks between your computer and HashiCorp's distribution channel.

However, bear in mind that `terraform init` by default stores only hashes for the platform on which it is running (e.g. only for Windows). If you expect the configuration to be used on other platforms, you can use the `terraform providers lock` command that we will learn more about in the *Generating one Terraform lock file with Windows and Linux compatibility* recipe in *Chapter 6*, *Applying a Basic Terraform Workflow*.

How it works...

In the first step, we run the `terraform init` command, which does the following:

- Downloads the specified version of the `azurerm` provider
- Creates a new file, `.terraform.lock.hcl`, that contains the installed provider version and hashes

Then we upgrade the version of the `azurerm` provider to version 3.0.0 and re-run the `init` command, which compares the new version hashed with the hashes written in `.terraform.lock.hcl`.

Due to the different version hashes, the `terraform init` command output (show in *Figure 1.27*) displays a version incompatibility error message that indicates how to upgrade.

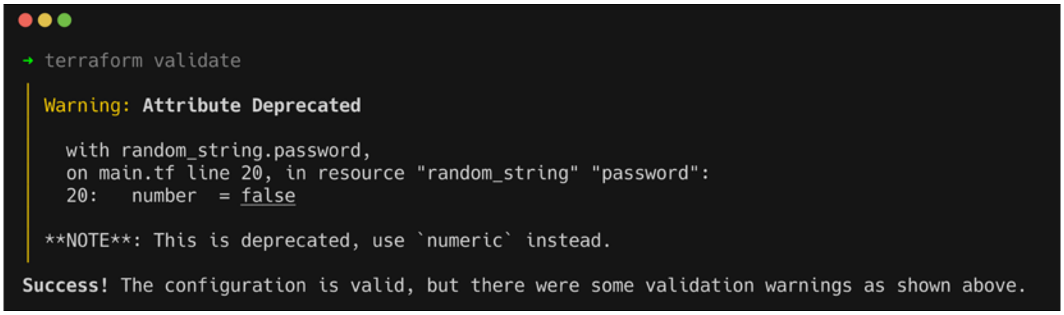
Finally, to fix this error, we run the `terraform init -upgrade` command, which upgrades `terraform.lock.hcl` with the new version's hashes.

There's more...

The main concept in this recipe is the `.terraform.lock.hcl` file, also called the **dependency file**, that contains all the information about the provider versions. It allows you to have the same provider versions on all workstations or CI/CD pipelines that apply this Terraform configuration.

There is another important consideration during the process of upgrading providers: before upgrading the provider, fix all deprecated Terraform resources that are upgraded with the new version of the provider.

Here is an example of deprecated attributes with the `random` provider:



```
→ terraform validate

  Warning: Attribute Deprecated

    with random_string.password,
    on main.tf line 20, in resource "random_string" "password":
    20:    number   = false

  **NOTE**: This is deprecated, use `numeric` instead.

Success! The configuration is valid, but there were some validation warnings as shown above.
```

Figure 1.24: Terraform validate command with depreciation

After fixing this deprecated attribute, the Terraform configuration will be fully valid.

See also

- The complete documentation about the Terraform dependency lock file is here: `https://www.terraform.io/language/files/dependency-lock`
- The official documentation about the Terraform provider upgrade is here: `https://learn.hashicorp.com/tutorials/terraform/provider-versioning`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

2

Writing Terraform Configurations

When you start writing Terraform configurations, you will notice very quickly that the language provided by Terraform is very rich and allows for a lot of manipulation.

In the recipes in this chapter, you will learn how to use the Terraform language effectively to apply it to real-life business scenarios. We will discuss how to use providers by specifying their version, and adding aliases to use multiple instances of the same provider. Then we will discuss how to make the code more dynamic with variables and outputs, and we will consider the use of built-in functions and conditions.

Finally, we will learn how to add dependencies between resources, add custom checks with pre- and postconditions, and check the provisioned infrastructure.

All the code examples explained in this book are for illustrative purposes only. Their purpose is to provision cloud infrastructure resources, which may have a cost depending on the cloud used. I strongly suggest that you delete these resources either manually or via the terraform destroy command, which we'll look at in detail in the recipe Destroying infrastructure resources in *Chapter 6*, *Applying a Basic Terraform Workflow*. Additionally, in a lot of the code on the GitHub repository of this chapter, you'll see the use of **random** resources, which allow you to have unique resources.

In this chapter, we will cover the following recipes:

- Configuring Terraform and the provider version to use
- Adding alias to a provider to use multiple instances of the same provider
- Manipulating variables

- Keeping sensitive variables safe
- Using local variables for custom functions
- Using outputs to expose Terraform provisioned data
- Calling Terraform's built-in functions
- Using YAML files in Terraform configuration
- Writing conditional expressions
- Generating passwords with Terraform
- Managing Terraform resource dependencies
- Adding custom pre- and postconditions
- Using checks for infrastructure validation

Let's get started!

Technical requirements

For this chapter, you need to have the Terraform binary installed on your computer. The source code for this chapter is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02`.

Configuring Terraform and the provider version to use

The default behavior of Terraform is that, when executing the `terraform init` command, the version of the Terraform binary (also called the **Command-Line Interface** (**CLI**)) used is the one installed on the local workstation. In addition, this command downloads the latest version of the providers used in the code.

Also, as we learned in *Chapter 1*, *Setting Up the Terraform Environment*, in the *Upgrading Terraform providers* recipe, this command creates the Terraform dependencies file, `.terraform.lock.hcl`.

However, for compatibility reasons, it is always advisable to avoid surprises so that you can specify which version of the Terraform binary is going to be used in the Terraform configuration. The following are some examples:

- A Terraform configuration that uses language constructs introduced in version 0.12 must be executed with that or a greater version

- A Terraform configuration that contains new features, such as `count` and `for_each`, in modules must be executed with Terraform version 0.13 or greater

For more details about the HCL syntax, read the documentation at `https://www.terraform.io/docs/configuration/syntax.html`.

In the same way and for the same reasons of compatibility, we may want to specify the provider version to be used.

In this recipe, we will learn how to specify the Terraform version, as well as the provider version, that will be used.

Getting ready

To start this recipe, we will write a basic Terraform configuration file that contains the following code:

```
variable "resource_group_name" {
  default = "rg_test"
}
resource "azurerm_resource_group" "rg" {
  name     = var.resource_group_name
  location = "westeurope"
}
resource "azurerm_public_ip" "pip" {
  name                         = "bookip"
  location                     = "westeurope"
  resource_group_name          = azurerm_resource_group.rg.name
  public_ip_address_allocation = "Dynamic"
  domain_name_label            = "bookdevops"
}
```

The source code of this Terraform configuration is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP02/version/specific-version.tf`.

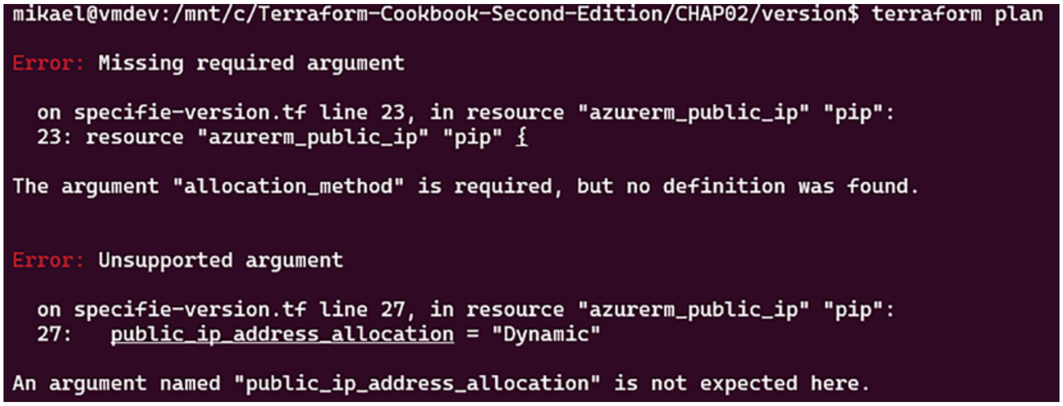
This example code provides resources in Azure (a resource group and a public IP address).

For more details about the Terraform `azurerm` provider, read the following documentation: `https://registry.terraform.io/providers/hashicorp/azurerm`.

This Terraform configuration contains the improvements that were made to the HCL 2.0 language since Terraform 0.12 using the new interpolation syntax.

For more details about these HCL enhancements, go to `https://www.slideshare.net/mitchp/terraform-012-deep-dive-hcl-20-for-infrastructure-as-code-remote-plan-apply-125837028`.

Finally, when executing the `terraform plan` command with this configuration, we get the following error messages:



```
mikael@vmdev:/mnt/c/Terraform-Cookbook-Second-Edition/CHAP02/version$ terraform plan

Error: Missing required argument

  on specifie-version.tf line 23, in resource "azurerm_public_ip" "pip":
  23: resource "azurerm_public_ip" "pip" {

The argument "allocation_method" is required, but no definition was found.


Error: Unsupported argument

  on specifie-version.tf line 27, in resource "azurerm_public_ip" "pip":
  27:   public_ip_address_allocation = "Dynamic"

An argument named "public_ip_address_allocation" is not expected here.
```

Figure 2.1: A Terraform plan without a specified version

This means that, currently, this Terraform configuration is not compatible with the latest version of the provider (version 2.56).

Now, we need to be aware of the following compliances:

- This configuration can only be executed if Terraform 0.13 (or higher) is installed on the local workstation.
- Our current configuration can be executed even if the `azurerm` provider evolves with breaking changes.

Regarding the new features provided by Terraform 0.13, read the change log at `https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md` and the upgrade guide at `https://developer.hashicorp.com/terraform/language/v1.1.x/upgrade-guides/0-13`.

The source code of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/version`.

How to do it...

First, we specify the Terraform version to be installed on the local workstation:

1. In the Terraform configuration, add the following block:

```
terraform {
  required_version = ">= 0.13,<=1"
}
```

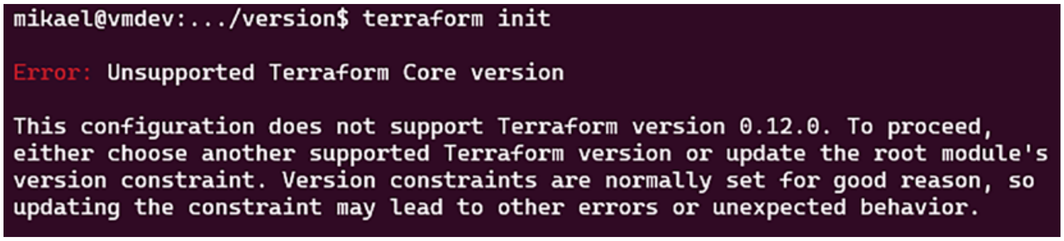
2. To specify the provider source and version to use, we need to add the `required_provider` block inside the same `terraform` block configuration:

```
terraform {
  ...
  required_providers {
    azurerm = {
      version = "2.10.0"
     }
  }
}
```

How it works...

When executing the `terraform init` command, Terraform will check that the version of the installed Terraform binary that executes the Terraform configuration corresponds to the version specified in the `required_version` property of the `terraform` block.

If it matches, it won't throw an error as it is greater than version 0.13. Otherwise, it will throw an error:

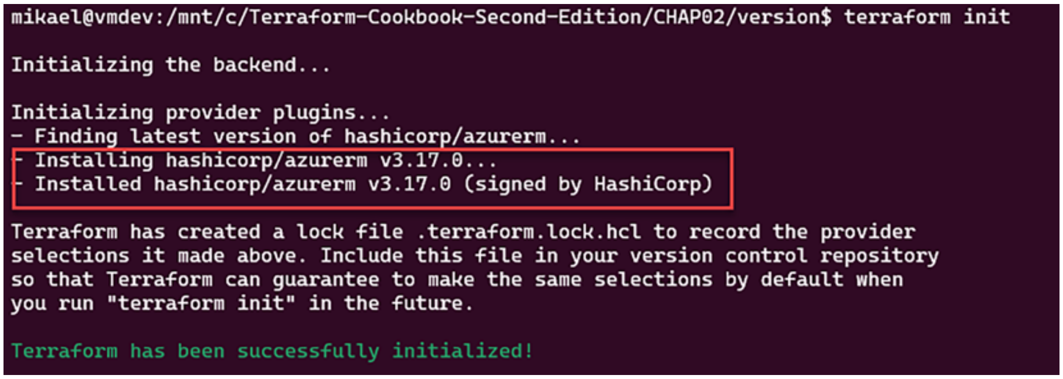


```
mikael@vmdev:.../version$ terraform init

Error: Unsupported Terraform Core version

This configuration does not support Terraform version 0.12.0. To proceed,
either choose another supported Terraform version or update the root module's
version constraint. Version constraints are normally set for good reason, so
updating the constraint may lead to other errors or unexpected behavior.
```

Figure 2.2: Terraform version incompatibility

Regarding the specification of the provider version, when executing the `terraform init` command, if no version is specified, Terraform downloads the latest version of the provider. Otherwise, it downloads the specified version, as shown in the following two screenshots.

The following screenshot shows the provider plugin that will be downloaded from the specified `source` without us specifying the required `version` (at the time of writing, the latest version of the provider is 3.17.0):

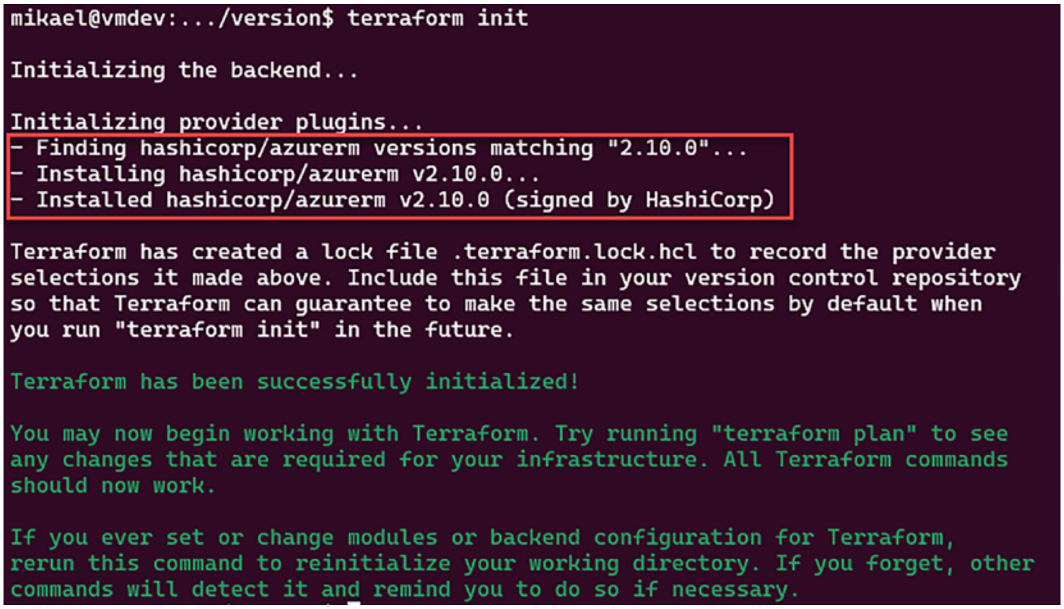


```
mikael@vmdev:/mnt/c/Terraform-Cookbook-Second-Edition/CHAP02/version$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/azurerm...
- Installing hashicorp/azurerm v3.17.0...
- Installed hashicorp/azurerm v3.17.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!
```

Figure 2.3: Terraform init downloads the latest version of the provider

As we can see, the specific version of the `azurerm` provider (3.17.0) has been downloaded.

In addition, the following screenshot shows the `azurerm` provider plugin that will be downloaded when we specify the required version (2.10.0):



```
mikael@vmdev:.../version$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "2.10.0"...
- Installing hashicorp/azurerm v2.10.0...
- Installed hashicorp/azurerm v2.10.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Figure 2.4: Terraform init downloads the specified provider version

As we can see, the specified version of the `azurerm` provider (2.10.0) has been downloaded.

For more details about the `required_version` block and provider versions, go to `https://www.terraform.io/docs/configuration/terraform.html#specifying-required-provider-versions`.

In the `required_version` block, we also add the `source` property, which was introduced in version 0.13 of Terraform and is documented at `https://www.terraform.io/language/upgrade-guides/0-13#explicit-provider-source-locations`.

There's more...

In this recipe, we learned how Terraform downloads the `azurerm` provider in several ways. What we did here applies to all providers you may wish to download.

It is also important to mention that the version of the Terraform binary that will be used is specified in the Terraform state file. This is to ensure that nobody applies this Terraform configuration with a lower version of the Terraform binary, thus ensuring that the format of the Terraform state file conforms with the correct version of the Terraform binary.

In the next recipe, we will implement a provider alias to use multiple instances of the same provider.

See also

- For more information about the properties of the Terraform block, go to `https://www.terraform.io/language/settings`.
- For more information about the properties of the providers, go to `https://www.terraform.io/language/providers/configuration`.
- More information about Terraform binary versioning is documented at `https://www.terraform.io/plugin/sdkv2/best-practices/versioning`.

Adding alias to a provider to use multiple instances of the same provider

When we write Terraform configuration, some providers contain properties for resource access and authentication such as a URL, authentication token, username, or password.

If we want to use multiple different configurations of the same provider in one Terraform configuration, for example, to provision resources in multiple Azure subscriptions in the same configuration, we can use the `alias` provider property.

Let's get started!

Getting ready

First, apply this basic Terraform code to create resources on Azure:

```
provider "azurerm" {
  subscription_id = "xxxx-xxx-xxx-xxxxxx"
  features {}
}

resource "azurerm_resource_group" "rg" {
  name     = "rg-sub1"
  location = "westeurope"
}

resource "azurerm_resource_group" "rg2" {
  name     = "rg-sub2"
  location = westeurope"
}
```

This Terraform configuration will create two Azure resource groups on the subscription that is configured by the provider (or in the default subscription on your Azure account).

In order to create an Azure resource group in another subscription, we need to use the `alias` property.

In this recipe, we will use the `alias` provider property, and to illustrate it we will provision two Azure resource groups in two different subscriptions in one Terraform configuration.

The requirement for this recipe is to have an Azure account, which you can get for free here: `https://azure.microsoft.com/en-us/free/`

We will also use the `azurerm` provider with basic configuration.

You can find your available active subscriptions (subscription IDs) at `https://portal.azure.com/#view/Microsoft_Azure_Billing/SubscriptionsBlade`.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/alias`

How to do it...

Perform the following steps to use multiple instances from one provider:

1. In `main.tf`, update the initial Terraform configuration in the `provider` section:

```
provider "azurerm" {
  subscription_id = "xxxx-xxx-xxxxx-xxxxxx"
  alias = "sub1"
  features {}
}

provider "azurerm" {
  subscription_id = "yyyy-yyyyy-yyyy-yyyyy"
  alias = "sub2"
  features {}
}
```

2. Then update the two existing `azurerm_resource_group` resources:

```
resource "azurerm_resource_group" "example1" {
  provider = azurerm.sub1
  name     = "rg-sub1"
  location = "westeurope"
}

resource "azurerm_resource_group" "example2" {
  provider = azurerm.sub2
  name     = "rg-sub2"
  location = "westeurope"
}
```

3. Finally, to apply the changes, run the Terraform workflow with the `init`, `plan`, and `apply` commands.

How it works...

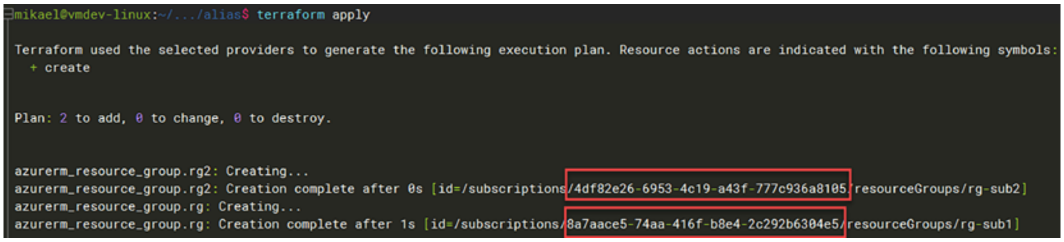
In *Step 1*, we duplicate the provider (`azurerm`) block and, on each provider, we add the `alias` property with an identification name. The first is `sub1` and the second is `sub2`.

Then we add the different `subscription_id` properties to specify the subscription where the resource will be created.

In *Step 2*, in each `azurerm_resource_group` resource, we add the `provider` property with a value that corresponds to that of the `alias` of the desired provider.

Each `azurerm_resource_group` resource targets the subscription using the provider's `alias`.

Finally, we run the `terraform init`, `plan` and `apply` commands. The screenshot below shows the `terraform apply` command:



```
mikael@vmdev-linux:~/.../alias$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create


Plan: 2 to add, 0 to change, 0 to destroy.


azurerm_resource_group.rg2: Creating...
azurerm_resource_group.rg2: Creation complete after 0s [id=/subscriptions/4df82e26-6953-4c19-a43f-777c936a8105/resourceGroups/rg-sub2]
azurerm_resource_group.rg: Creating...
azurerm_resource_group.rg: Creation complete after 1s [id=/subscriptions/8a7aace5-74aa-416f-b8e4-2c292b6304e5/resourceGroups/rg-sub1]
```

Figure 2.5: Running the apply command

We can see the two different subscriptions where the Azure resource group will be created.

See also

- The documentation of the provider `alias` is available at `https://www.terraform.io/language/providers/configuration#alias-multiple-provider-configurations`.

- Here's a great article on using the provider `alias`: `https://build5nines.com/terraform-deploy-to-multiple-azure-subscriptions-in-single-project/`.

Manipulating variables

When you write a Terraform configuration where all the properties are hardcoded in the code, you often find yourself faced with the problem of having to duplicate it to reuse it.

In this recipe, we'll learn how to make the Terraform configuration more dynamic by using variables.

Getting ready

To begin, we are going to work on the `main.tf` file, which contains a basic Terraform configuration:

```
resource "azurerm_resource_group" "rg" {
  name     = "My-RG"
  location = "West Europe"
}
```

As we can see, the `name` and `location` properties have values statically written in the code.

Let's learn how to make them dynamic using variables.

The source code of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/variables`.

How to do it…

Perform the following steps:

1. In the same `main.tf` file, add the following variable declarations:

```
variable "resource_group_name" {
  description ="The name of the resource group"
}
variable "location" {
  description ="The name of the Azure region"
  default ="westeurope"
}
```

2. Then, modify the Terraform configuration we had at the beginning of this recipe so that it refers to our new variables, as follows:

```
resource "azurerm_resource_group" "rg" {
  name     = var.resource_group_name
  location = var.location
}
```

3. Finally, in the same folder that contains the `main.tf` file, create a new file called `terraform.tfvars` and add the following content:

```
resource_group_name = "My-RG"
location            = "westeurope"
```

How it works...

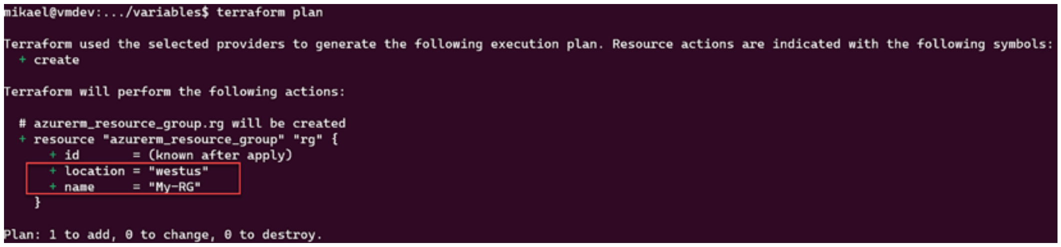
In *Step 1*, we wrote the declaration of the two variables, which consists of the following elements:

- **A variable name**: This must be unique to this Terraform configuration and must be explicit enough to be understood by all the contributors of the code.
- **A description of what this variable represents**: This description is optional but is recommended because it can be displayed by the CLI and can also be integrated into the documentation, which is automatically generated.
- **A default value**: This is optional. Not setting a default value makes it mandatory to enter a value.

Then, in *Step 2*, we modified the Terraform configuration to use these two variables. We did this using the `var.<name of the variable>` syntax.

Finally, in *Step 3*, we set values for the variables in the `terraform.tfvars` file, which is used natively by Terraform.

The result of executing this Terraform configuration is shown in the following screenshot:



```
mikael@vmdev:.../variables$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id       = (known after apply)
      + location = "westus"
      + name     = "My-RG"
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Figure 2.6: Using the terraform.tfvars file

There's more...

Setting a value for the variable is optional in the `terraform.tfvars` file since we have set a default value for the variable.

Apart from this `terraform.tfvars` file, it is possible to give a variable a value using the `-var` option of the `terraform plan` and `terraform apply` commands, as shown in the following command:

```
terraform plan -var "location=westus"
```

So, with this command, the `location` variable declared in our code will have a value of `westus` instead of `westeurope`.

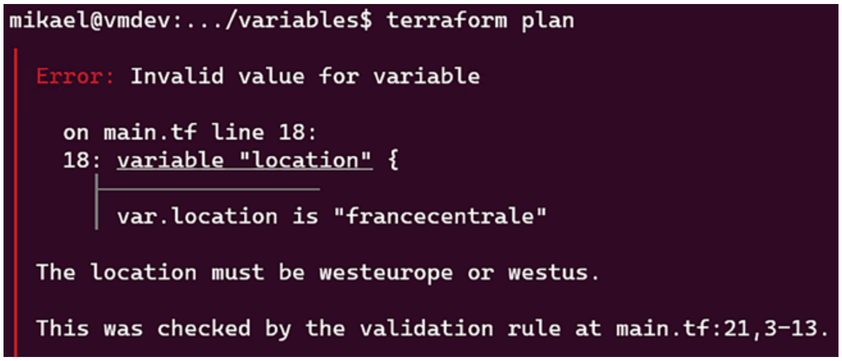
In addition, in the 0.13 version of Terraform, we can now create custom validation rules for variables, which makes it possible for us to verify a value during the `terraform plan` execution.

In our recipe, we can complete the `location` variable with a validation rule in the `validation` block as shown in the following code:

```
variable "location" {
  description ="The name of the Azure location"
  default ="westeurope"
  validation {
    condition = contains(["westeurope","westus"], var.location)
error_message = "The location must be westeurope or westus."
  }
}
```

In the preceding configuration, the rule checks if the value of the `location` variable is `westeurope` or `westus`.

If you put in an invalid value for the `location` variable, such as `francecentrale`, the validation rule will display **The location must be westeurope or westus**:



```
mikael@vmdev:.../variables$ terraform plan

 Error: Invalid value for variable

   on main.tf line 18:
   18: variable "location" {
     ├────────────────
     │ var.location is "francecentrale"

 The location must be westeurope or westus.

 This was checked by the validation rule at main.tf:21,3-13.
```

Figure 2.7: Variable validation

For more information about variable custom rules validation, read the documentation at `https://www.terraform.io/docs/configuration/variables.html#custom-validation-rules`.

Finally, there is another alternative to setting a value to a variable, which consists of setting an environment variable called `TF_VAR_<variable name>`. In our case, we can create an environment variable called `TF_VAR_location` with a value of `westus` and then execute the `terraform plan` command in the classical way.

Note that using the `-var` option or the `TF_VAR_<name of the variable>` environment variable doesn't hardcode these variables' values inside the Terraform configuration. Terraform makes it possible for us to give values of variables on the fly. But be careful – these options can have consequences if the same code is executed with other values initially provided in parameters and if the plan's output isn't reviewed carefully.

See also

In this recipe, we looked at the basic use of variables. We will learn how to protect sensitive variables in the next recipe, and we will examine more advanced uses of these variables when we learn how to manage environments in *Chapter 3*, *Scaling Your Infrastructure with Terraform*, in the *Managing infrastructure in multiple environments* recipe.

For more information on Terraform variables, refer to the documentation at `https://www.terraform.io/docs/configuration/variables.html`.

Keeping sensitive variables safe

In the *Manipulating variables* recipe in this chapter, we learned how to use variables to make our Terraform configuration more dynamic. By default, all variables' values used in the configuration will be stored in clear text in the Terraform state file, and this value will also be in clear text in the console output execution.

In this recipe, we will learn how to keep Terraform variable information safe from prying eyes by not displaying their values in clear text in the console output.

Getting ready

To start with, we will use the Terraform configuration available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/sample-app`, which provisions an Azure Web App with custom app settings.

To illustrate this, we will add a custom application API key, which has a sensitive value, in the key-value app setting.

The source code of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/sample-app`.

How to do it...

Perform the following steps:

1. In the `main.tf` file, which contains our Terraform configuration, in `azurerm_linux_web_app` we will add an `app_settings` property that is set with the `api_key` variable:

```
resource "azurerm_linux_web_app" "app" {
  name                = "${var.app_name}-${var.environment}
-${random_string.random.result}”
  location            = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id     = azurerm_service_plan.plan-app.id

  site_config {}
  app_settings = {
API_KEY = var.api_key
  }
        }
```

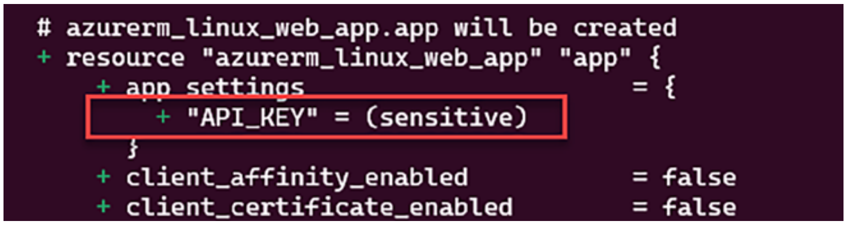
2. Then, in `variable.tf`, declare the `custom_app_settings` variable:

```
variable "api_key " {
  description = "Custom application api key"
  sensitive = true
}
```

3. In `terraform.tfvars`, we instantiate this variable with these values (as an example for this book):

```
api_key = "xxxxxxxxxxxxxxxxx"
```

4. Finally, we run the `terraform plan` command. The following screenshot shows a part of this execution:



```
  # azurerm_linux_web_app.app will be created
  + resource "azurerm_linux_web_app" "app" {
      + app_settings                            = {
          + "API_KEY" = (sensitive)
        }
      + client_affinity_enabled                 = false
      + client_certificate_enabled              = false
```

Figure 2.8: Terraform doesn't display the sensitive variable value

5. We can see that the value of the `api_key` property (which is in uppercase in the Terraform plan output) in `app_settings` isn't displayed in the console output.

How it works...

In the recipe, we add the `sensitive` flag to the `api_key` variable, which enables the protection of the variable. By enabling this flag, the `terraform plan` command (and the `apply` command) doesn't display the value of the variable in the console output in clear text.

There's more...

Be careful! The `sensitive` variable property protects the value of this variable from being displayed in the console output, but the value of this is still written in clear text in the Terraform state file.

Then, if this Terraform configuration is stored in a source control such as Git, the default value of this variable or the `tfvars` file is also readable as clear text in the source code.

So, to protect the value of this variable in source control, we can set the value of this variable by using Terraform's environment variable technique with the format `TF_VAR_<variable name>` as we learned in the previous recipe, *Manipulating variables*.

In our scenario, we can set the `TF_VAR_api_key = "xxxxxxxxxxx"` environment variable just before the execution of the `terraform plan` command.

One scenario in which the sensitive variable is effective is when Terraform is executed in a CI/CD pipeline; then, unauthorized users can't read the value of the variable.

Finally, one best practice is to use an external secret management solution such as Azure Key Vault or HashiCorp Vault (`https://www.vaultproject.io/`) to store your secrets and use Terraform providers to get these secrets' values.

See also

- Documentation for `sensitive` variables is available at `https://www.terraform.io/language/values/variables#suppressing-values-in-cli-output`.
- For more information on `sensitive` variables, read the Terraform tutorial at `https://learn.hashicorp.com/tutorials/terraform/sensitive-variables`.

Using local variables for custom functions

In the *Manipulating variables* recipe in this chapter, we learned how to use variables to dynamize our Terraform configuration. Sometimes, this can be a bit tedious when it comes to using combinations of variables.

In this recipe, we will learn how to implement local variables and use them as custom functions.

Getting ready

To start with, we will use the following Terraform configuration:

```
variable "application_name" {
  description = "The name of application"
}
variable "environment_name" {
  description = "The name of environment"
}
variable "country_code" {
  description = "The country code (FR-US-...)"
}
```

```
resource "azurerm_resource_group" "rg" {
  name = "XXXX" # VARIABLE TO USE
  location = "West Europe"
}
resource "azurerm_public_ip" "pip" {
  name = "XXXX" # VARIABLE TO USE
  location = "West Europe"
  resource_group_name = azurerm_resource_group.rg.name
  allocation_method = "Dynamic"
  domain_name_label = "mydomain"
}
```

The goal of this recipe is to consistently render the names of the Azure resources. We must provide them with the following nomenclature rule:

```
CodeAzureResource - Name Application - Environment name - Country Code
```

The source code of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/localvariables`.

How to do it…

Follow these steps:

1. In the `main.tf` file, which contains our Terraform configuration, we will add a local variable called `resource_name`, along with the following code:

   ```
   locals {
     resource_name = "${var.application_name}-${var.environment_name}-
   ${var.country_code}"
   }
   ```

2. We then use this local variable in the resources with the following code:

   ```
   resource "azurerm_resource_group" "rg" {
    name = "RG-${local.resource_name}"
    location = "westeurope"
   }
   resource "azurerm_public_ip" "pip" {
    name = "IP-${local.resource_name}"
    location = "westeurope"
    resource_group_name = azurerm_resource_group.rg.name
   ```

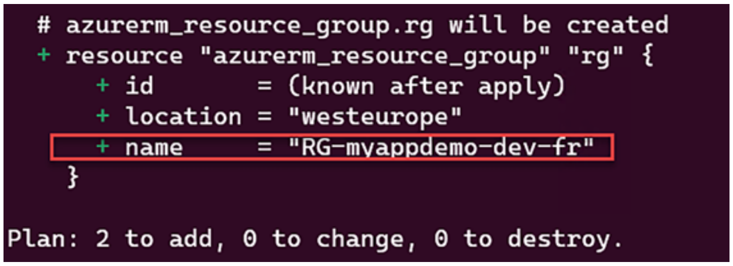
```
  allocation_method = "Dynamic"
  domain_name_label = "mydomain"
}
```

How it works...

In *Step 1*, we created a variable called `resource_name` that is local to our Terraform configuration. This allows us to create a combination of several Terraform variables (which we will see the result of in the *Using outputs to expose Terraform provisioned data* recipe of this chapter).

Then, in *Step 2*, we used this local variable with the `local.<name of the local variable>` expression. Moreover, in the `name` property, we used it as a concatenation of a variable and static text, which is why we used the `"${}"` syntax.

The result of executing this Terraform configuration is as follows:



```
  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "RG-myappdemo-dev-fr"
    }

Plan: 2 to add, 0 to change, 0 to destroy.
```

Figure 2.9: Using the locals variable

In the previous screenshot, we can see the output of executing the `terraform plan` command with the `name` of the resource group that we calculated with the `locals` variable.

There's more...

The difference between a local variable and Terraform variable is that the local variable can't be redefined in the Terraform variables file (`tfvars`), with environment variables, or with the `-var` CLI argument.

See also

- For more information on `local` block, look at the following documentation: `https://www.terraform.io/docs/configuration/locals.html`.
- The Terraform locals learning lab is available at `https://learn.hashicorp.com/tutorials/terraform/locals`.

Using outputs to expose Terraform provisioned data

When using Infrastructure as Code tools such as Terraform, it is often necessary to retrieve output values from the provisioned resources after code execution.

One of the uses of these output values is that they can be used after execution by other Terraform configurations or external programs. This is often the case when the execution of the Terraform configuration is integrated into a CI/CD pipeline.

For example, we can use these output values in a CI/CD pipeline that creates an Azure App Service instance with Terraform and deploys the application to this Azure App Service instance. In this example, we can have the name of the Azure App Service instance as the output of the Terraform configuration. These output values are also very useful for transmitting information through modules, which we will see in detail in *Chapter 5*, *Managing Terraform State*.

In this recipe, we will learn how to implement output values in Terraform configuration to get the name of the provisioned Azure Web App in the output.

Getting ready

To proceed, we are going to add some Terraform configuration that we already have in the existing `main.tf` file.

The following is an extract of this existing code, which provides an app service in Azure:

```
...
resource "azurerm_linux_web_app" "app" {
  name                = "${var.app_name}-${var.environment}"
  location            = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id     = azurerm_service_plan.plan-app.id
  site_config {}
}
...
```

The source code of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/sample-app`.

How to do it...

To ensure we have an output value, we will just add the following code to the `main.tf` file:

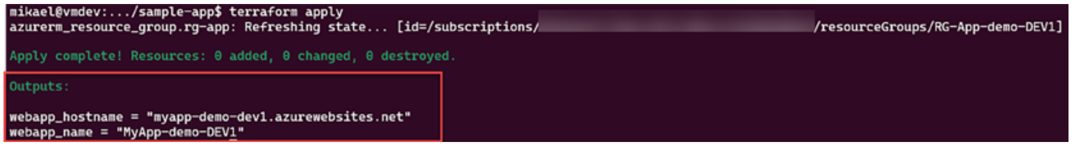
```
output "webapp_name" {
  description = "Name of the webapp"
  value = azurerm_linux_web_app.app.name
}
```

How it works...

The **output** block of Terraform is defined by a name, `webapp_name`, and a value, `azurerm_linux_web_app.app.name`. These refer to the name of the Azure App Service instance that is provided in the same Terraform configuration. Optionally, we can add a `description` property that describes what the output returns, which can also be very useful for autogenerated documentation.

It is, of course, possible to define more than one output in the same Terraform configuration.

The outputs are stored in the Terraform state file and are displayed when the `terraform apply` command is executed, as shown in the following screenshot:



```
mikael@vmdev:.../sample-app$ terraform apply
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/[redacted]/resourceGroups/RG-App-demo-DEV1]

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

webapp_hostname = "myapp-demo-dev1.azurewebsites.net"
webapp_name = "MyApp-demo-DEV1"
```

Figure 2.10: Terraform outputs

Here, we see two output values that are displayed at the end of the execution.

There's more...

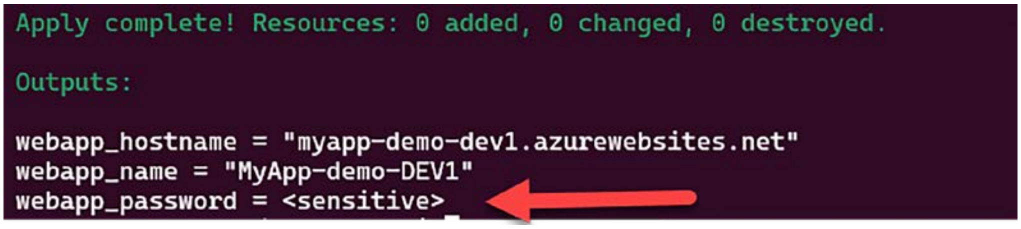
There are two ways to retrieve the values of the output to use them:

- By using the `terraform output` command in the Terraform CLI, which we will see in *Chapter 6*, *Applying a Basic Terraform Workflow*, in the *Exporting the output in JSON* recipe
- By using the `terraform_remote_state` data source, which we will discuss in *Using external resources from other state files* recipe

Another consideration is that we can also expose an output that contains sensitive values with the goal of avoiding displaying their values in clear text in the console output. To do this, add the `sensitive = true` property to the output. The code below creates the output for the Azure App Service password:

```
output "webapp_password" {
  description = "credential of the webapp"
  value       = azurerm_linux_web_app.app.site_credential
  sensitive = true
}
```

The image below shows the execution of the Terraform configuration that contains this output:



```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

webapp_hostname = "myapp-demo-dev1.azurewebsites.net"
webapp_name = "MyApp-demo-DEV1"
webapp_password = <sensitive>
```

Figure 2.11: Sensitive output

We can see that the value of the `webapp_password` output isn't displayed in the console.

Be careful; the output value will still be in clear text in the Terraform state file.

See also

- Documentation on Terraform outputs is available at `https://www.terraform.io/docs/configuration/outputs.html`.

Calling Terraform's built-in functions

When provisioning infrastructure or handling resources with Terraform, it is sometimes necessary to use transformations or combinations of elements provided in the Terraform configuration.

For this purpose, the language supplied with Terraform includes functions that are built in and can be used in any Terraform configuration.

In this recipe, we will discuss how to use built-in functions to apply transformations to code.

Getting ready

To complete this recipe, we will start from scratch regarding the Terraform configuration, which will be used to provision a resource group in Azure. This resource group will be named according to the following naming convention:

```
RG-<APP NAME>-<ENVIRONMENT>
```

The result of this transformation will return a name that should be entirely in uppercase.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/fct`.

How to do it...

Perform the following steps:

1. In a new local folder, create a file called `main.tf`.
2. In this `main.tf` file, write the following code:

```
variable "app_name" {
  description = "Name of application"
}
variable "environement" {
  description = "Environement Name"
}
```

3. Finally, in this `main.tf` file, write the following Terraform configuration:

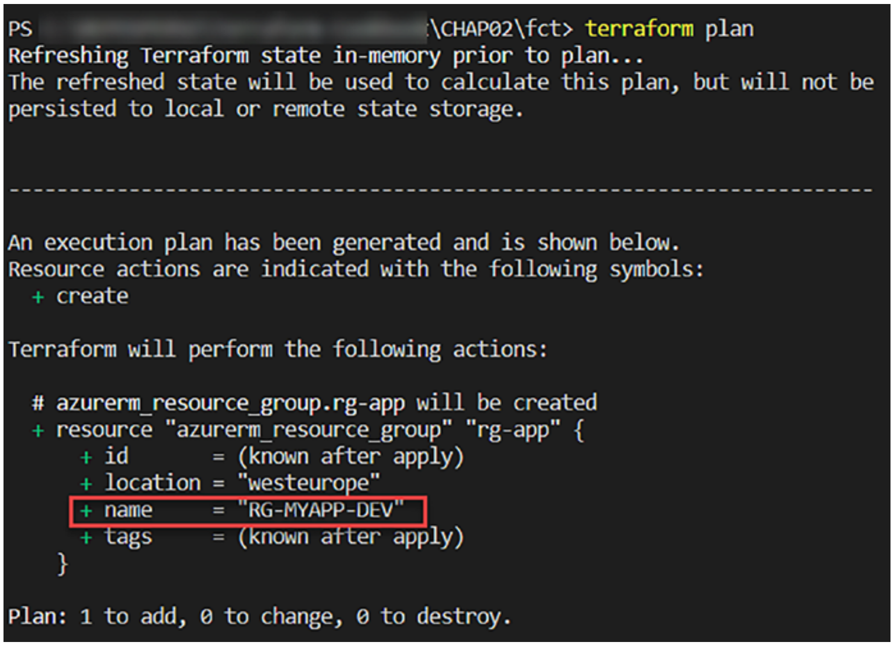
```
resource "azurerm_resource_group" "rg-app" {
  name     = upper(format("RG-%s-%s",var.app-name,var.environement))
  location = "westeurope"
}
```

How it works...

In *Step 3*, we defined the property name of the resource with a Terraform `format` function, which allows us to format text. In this function, we used the `%s` verb to indicate that it is a character string that will be replaced, in order, by the name of the application and the name of the environment.

Furthermore, to capitalize everything inside, we encapsulate the `format` function in the upper function, which capitalizes all its contents.

The result of executing these Terraform commands on this code can be seen in the following screenshot:



```
PS [redacted]\CHAP02\fct> terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.rg-app will be created
  + resource "azurerm_resource_group" "rg-app" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "RG-MYAPP-DEV"
      + tags     = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Figure 2.12: Terraform built-in function to capitalize text

Thus, thanks to these functions, it is possible to control the properties that will be used in the Terraform configuration. This also allows us to apply transformations automatically, without having to impose constraints on the user using the Terraform configuration.

See also

There are a multitude of predefined functions in Terraform. The full list can be found at `https://www.terraform.io/docs/configuration/functions.html` (navigate with the left menu to see all built-in functions):

- To read more details about the `format` function, refer to the documentation at `https://www.terraform.io/language/functions/format`.
- To read more details about the upper function, refer to the documentation at `https://www.terraform.io/language/functions/upper`.

Using YAML files in Terraform configuration

In the previous recipes, we learned that to set dynamic values inside Terraform configuration, we can use variables.

In some use cases, we must use an external source for configuration, such as **JSON** or **YAML** files, and we can imagine that these files are provided manually by external teams or generated automatically by external systems, and we can't rewrite these files in Terraform variables.

The goal of this recipe is to show how to use a YAML file inside a Terraform configuration.

Let's get started!

Getting ready

To complete this recipe, we have a YAML file named `network.yaml` with the following content:

```
vnet: "myvnet"
address_space: "10.0.0.0/16"
subnets:
- name: subnet1
  iprange: "10.0.1.0/24"
- name: subnet2
  iprange: "10.0.2.0/24"
```

This file contains the configuration of the Azure network with virtual network and subnets configuration, and it's placed in the same folder as the Terraform configuration.

In our Terraform configuration, we will use this YAML file to provision the Azure virtual network and subnets.

The source code of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/yaml`.

How to do it…

Perform the following steps:

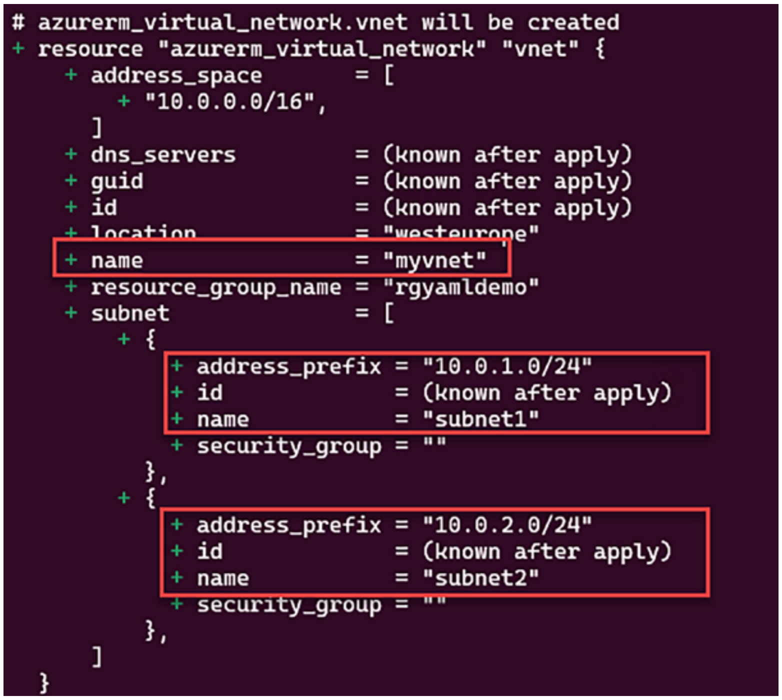
1. First, in `main.tf`, create a `locals` variable called `network` that calls the built-in `yamldecode` Terraform function:

   ```
   locals {
       network = yamldecode(file("network.yaml"))
   }
   ```

2. Then call this `local.network` variable and its subproperties, which are defined in the YAML file inside the Terraform resource:

```
resource "azurerm_virtual_network" "vnet" {
  name                = local.network.vnet
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  address_space       = [local.network.address_space]
  dynamic "subnet" {
    for_each = local.network.subnets
    content {
      name           = subnet.value.name
      address_prefix = subnet.value.iprange
    }
  }
}
```

3. Finally, run the Terraform workflow with the `init`, `plan`, and `apply` commands. The following picture shows the `plan` execution:



```
# azurerm_virtual_network.vnet will be created
+ resource "azurerm_virtual_network" "vnet" {
    + address_space       = [
        + "10.0.0.0/16",
      ]
    + dns_servers         = (known after apply)
    + guid                = (known after apply)
    + id                  = (known after apply)
    + location            = "westeurope"
    + name                = "myvnet"
    + resource_group_name = "rgyamldemo"
    + subnet              = [
        + {
            + address_prefix = "10.0.1.0/24"
            + id             = (known after apply)
            + name           = "subnet1"
            + security_group = ""
          },
        + {
            + address_prefix = "10.0.2.0/24"
            + id             = (known after apply)
            + name           = "subnet2"
            + security_group = ""
          },
      ]
  }
```

Figure 2.13: Terraform uses the YAML file

4. We can see that the Terraform configuration uses the content configuration of the YAML file.

How it works...

In *Step 1*, we use the built-in Terraform function `yamldecode`, which takes on the parameters of the `network.yaml` file. This function decodes YAML content to Terraform key-value maps.

The result of this map is stored in a local variable named `network`.

Then in *Step 2*, we call this local variable by using `local.network` and use all the sub-keys as object notation, defined in the YAML configuration.

That is all for the Terraform configuration. Finally, we run the `terraform init`, `plan`, and `apply` Terraform commands.

The `plan` execution shows that Terraform uses YAML for configuration.

There's more...

In this recipe, we saw how to decode a YAML file, but we can also encode a YAML file, from Terraform to YAML, by using the built-in `yamlencode` Terraform function (`https://www.terraform.io/language/functions/yamlencode`).

We learned an example of decoding a YAML file. With Terraform we can do the same operations with a JSON file using the built-in `jsondecode` and `jsonencode` functions.

However, it is better to use Terraform variables for Terraform variable validation by using the `terraform validate` command. Indeed, the YAML file will not be integrated into the validation of Terraform if it is badly formatted or if some information is missing – in these instances it will throw an error.

See also

- The documentation of the built-in `yamldecode` function is available at `https://www.terraform.io/language/functions/yamldecode`.
- The built-in `jsonencode` function documentation is available at `https://www.terraform.io/language/functions/jsonencode`.
- The built-in `jsondecode` function documentation is available at `https://www.terraform.io/language/functions/jsondecode`.

Writing conditional expressions

When writing the Terraform configuration, we may need to make the code more dynamic by integrating various conditions. In this recipe, we will discuss an example of a conditional expression.

Getting ready

For this recipe, we will use the Terraform configuration we wrote in the previous recipe, the code for which is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/fct`.

We will complete this code by adding a condition to the name of the resource group. This condition is as follows: if the name of the environment is equal to `Production`, then the name of the resource group will be in the form `RG-<APP NAME>`; otherwise, the name of the resource group will be in the form `RG-<APP NAME>-<ENVIRONMENT NAME>`.

How to do it…

In the Terraform configuration of the `main.tf` file, modify the code of the resource group as follows:

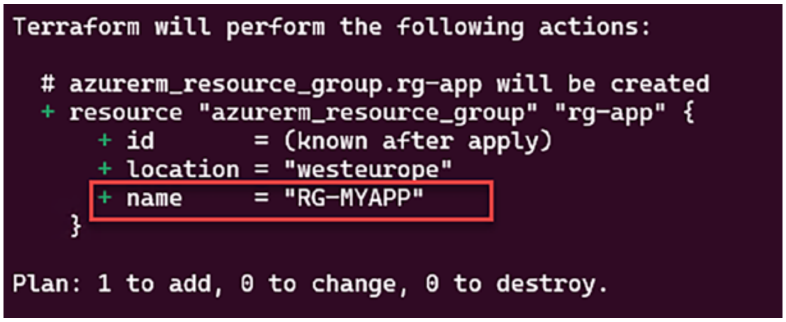
```
resource "azurerm_resource_group" "rg-app" {
    name = var.environment == "Production" ? upper(format("RG-%s",var.app-
name)) : upper(format("RG-%s-%s",var.app-name,var.environment))
    location = "westeurope"
}
```

How it works…

Here, we added the following condition:

```
condition ? true assert : false assert
```

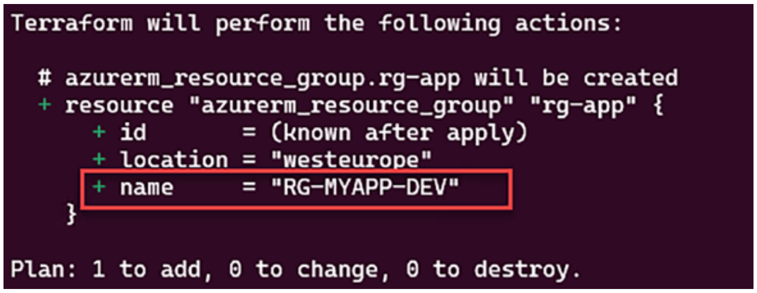
The result of executing Terraform commands on this code if the `environment` variable is equal to `Production` can be seen in the following screenshot:



```
Terraform will perform the following actions:

  # azurerm_resource_group.rg-app will be created
  + resource "azurerm_resource_group" "rg-app" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "RG-MYAPP"
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Figure 2.14: Conditional expression first use case

If the environment variable is not equal to Development, we'll get the following output:



```
Terraform will perform the following actions:

  # azurerm_resource_group.rg-app will be created
  + resource "azurerm_resource_group" "rg-app" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "RG-MYAPP-DEV"
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Figure 2.15: Conditional expression second use case

There's more...

One of the usual use cases for conditional expressions is to implement a pattern of `features` flags.

With `features` flags, we can make the provisioning of resources optional in a dynamic way, as shown in the following code snippet:

```
resource "azurerm_application_insights" "appinsight-app" {
  count = var.use_appinsight == true ? 1 : 0
  ....
}
```

In this code, we have indicated to Terraform that if the `use_appinsight` variable is `true`, then the `count` property is `1`, which will allow us to provision one Azure Application Insights resource. Conversely, where the `use_appinsight` variable is `false`, the `count` property is `0` and in this case, Terraform does not create an Application Insights resource instance.

See also

Documentation on conditional expressions in Terraform can be found at `https://www.terraform.io/language/expressions/conditionals`.

Generating passwords with Terraform

When provisioning infrastructure with Terraform, there are some resources that require passwords in their properties, such as account credentials, VMs, and database connection strings.

To ensure better security by not writing passwords as plaintext in the configuration, you can use a `random` Terraform provider, which allows you to generate a random string that can be used as a password.

In this recipe, we will discuss how to generate a password with Terraform and assign it to a resource.

Getting ready

In this recipe, we need to provision a VM in Azure that will be provisioned with an administrator password generated dynamically by Terraform.

To do this, we will use an already existing Terraform configuration that provisions a VM in Azure.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/password`.

How to do it…

Perform the following steps:

1. In the Terraform configuration file for the VM, add the following code:

```
resource "random_password" "password" {
  length = 16
  special = true
  override_special = "_%@"
}
```

2. Then, in the code of the resource itself, modify the `password` property with the following code:

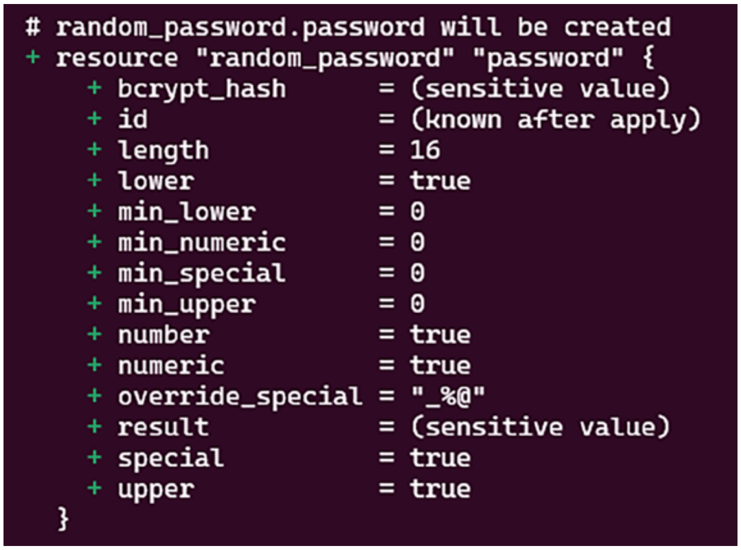
```
resource "azurerm_linux_virtual_machine" "myterraformvm" {
  name = "myVM"
  location = "westeurope"
  resource_group_name = azurerm_resource_group.myterraformgroup.name
     computer_name = "vmdemo"
     admin_username = "uservm"
     admin_password = random_password.password.result


....
}
```

How it works...

In *Step 1*, we added the Terraform `random_password` resource from the `random` provider, which allows us to generate strings according to the properties provided. These will be `sensitive`, meaning that they're treated as sensitive in the CLI by Terraform, so they will not be displayed to the user.

Then, in *Step 2*, we used its result (with the `result` property) in the `password` property of the VM.

The result of executing the `terraform plan` command on this code can be seen in the following screenshot:



```
# random_password.password will be created
+ resource "random_password" "password" {
    + bcrypt_hash      = (sensitive value)
    + id               = (known after apply)
    + length           = 16
    + lower            = true
    + min_lower        = 0
    + min_numeric      = 0
    + min_special      = 0
    + min_upper        = 0
    + number           = true
    + numeric          = true
    + override_special = "_%@"
    + result           = (sensitive value)
    + special          = true
    + upper            = true
  }
```

Figure 2.16: Generating a password with Terraform

As we can see, the result is `(sensitive value)`.

Please note that the fact a property is sensitive in Terraform means that it cannot be displayed when using the Terraform `plan` and `apply` commands in the console output display.

On the other hand, it will be present in clear text in the Terraform state file.

See also

- To find out more about the `random` provider, read the following documentation: `https://registry.terraform.io/providers/hashicorp/random/`.

- Documentation regarding sensitive data in Terraform state files is available at `https://www.terraform.io/docs/state/sensitive-data.html`.

Managing Terraform resource dependencies

One of Terraform's main features is to allow the parallelization of operations while considering resource dependencies.

In this recipe, we will learn how to create dependencies between resources. We will do this using both implicit and explicit dependencies.

Let's get started!

Getting ready

To start this recipe, we will use the following Terraform configuration to provision an Azure resource group and, inside it, one Azure virtual network.

Here is the basic configuration:

```
resource "azurerm_resource_group" "rg" {
  name     = "rgdep"
  location = "westeurope"
}

resource "azurerm_virtual_network" "vnet" {
  name                = "vnet"
  location            = "westeurope"
  resource_group_name = "rgdep"
  address_space       = ["10.0.0.0/16"]
}
```

The problem with the above configuration is that there are no Terraform dependencies between the resource group and the virtual network. Since Terraform executes its operations according to its dependency graph calculation, during the `apply` execution the virtual network can be created before the resource group. However, this isn't acceptable because the virtual network must be created after the resource group.

The goal of this recipe is to learn how to create dependencies between the Azure virtual network and the Azure resource group.

Find out more about the concept of dependencies at https://www.terraform.io/language/resources/behavior#resource-dependencies.

The source code of this recipe is available at https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/dep.

How to do it...

Perform the following steps:

1. To create implicit dependencies, update the azurerm_virtual_network resource with the following code:

```
resource "azurerm_virtual_network" "vnet" {
  name                = "vnet"
  location            = "westeurope"
  resource_group_name = azurerm_resource_group.rg.name
  address_space       = ["10.0.0.0/16"]
}
```

2. To create explicit dependencies, update the azurerm_virtual_network resource with the following code:

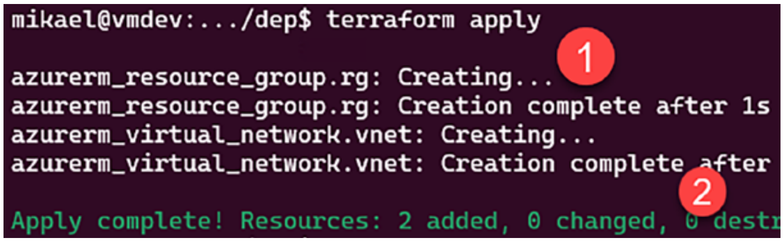
```
resource "azurerm_virtual_network" "vnet" {
  name                = "vnet"
  location            = "westeurope"
  resource_group_name = "rgdep"
  address_space       = ["10.0.0.0/16"]

  depends_on = [azurerm_resource_group.rg]
}
```

How it works...

In the first configuration, we use an **implicit** dependency. By using the resource_group_name = azurerm_resource_group.rg.name property, Terraform creates a dependency that is waiting for the creation of the resource so it can know its name and pass it to the virtual network.

In the second configuration, we use an **explicit** dependency by using the depends_on meta-argument, which contains the list of resources that must be created before this current resource. Here, we explicitly set that the Azure resource group must be created before creating the Azure virtual network.

Finally, we execute the `terraform apply` command. The following image shows that this execution has dependencies:



```
mikael@vmdev:.../dep$ terraform apply

azurerm_resource_group.rg: Creating...
azurerm_resource_group.rg: Creation complete after 1s
azurerm_virtual_network.vnet: Creating...
azurerm_virtual_network.vnet: Creation complete after

Apply complete! Resources: 2 added, 0 changed, 0 destr
```

Figure 2.17: Terraform uses implicit and explicit dependency

We can see the order of resource creation: the Azure resource group [1] is created before the Azure virtual network [2].

And when we run the `terraform destroy` command, the Azure virtual network is destroyed before the Azure resource group.

There's more...

If we have the choice between using an implicit or explicit dependency, it's recommended to use an implicit dependency as explained in this documentation: `https://www.terraform.io/language/meta-arguments/depends_on#processing-and-planning-consequences`

It is possible to display the Terraform resources graph's dependencies by using the `terraform graph` command, which we will learn about in detail in *Chapter 6*, *Applying a Basic Terraform Workflow*, in the *Generating the graph dependencies* recipe.

See also

- The Terraform tutorial on dependencies is available at `https://learn.hashicorp.com/tutorials/terraform/dependencies`.
- The documentation on the `depends_on` meta-argument is available at `https://www.terraform.io/language/meta-arguments/depends_on`.

Adding custom pre and postconditions

In a previous recipe, *Manipulating variables*, we learned that it is possible to add condition validation inside the variable definition.

In Terraform version 1.2 and newer, it's possible to add custom validation directly in resources, modules, or data sources with preconditions and postconditions.

These customs validations allow Terraform to set some custom rules during the execution of `terraform plan`. The precondition will be checked just before the rendering of the plan and the postcondition will be checked just after the rendering.

Let's get started!

Getting ready

To complete this recipe, we will start with this basic Terraform configuration:

```
resource "azurerm_virtual_network" "vnet" {
  name                = "vnet"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  address_space       =  [var.address_space]
}
```

The above Terraform configuration creates an Azure virtual network.

The first check that we want to perform is to be sure that the `address_space` variable's value is `IP mask /16`.

The second check is to verify that the region (`location`) of the virtual network is `"westeurope"`.

The source code of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/prepostcond`.

How to do it...

The following step shows you how to perform the first check, which is the verification of the IP address range:

1. Update the Terraform configuration of the Azure virtual network with the code below:

```
resource "azurerm_virtual_network" "vnet" {
…..
  address_space       =  [var.address_space]
  lifecycle {
    precondition {
      condition = cidrnetmask(var.address_space) == "255.255.0.0"
      error_message = "The IP Range must be /16"
    }
  }
}
```

2. Then, add the second check to verify the location by adding this configuration in the Azure virtual network:

```
resource "azurerm_virtual_network" "vnet" {
  name                = "vnet"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  address_space       =  [var.address_space]
  lifecycle {
    precondition {
      condition = cidrnetmask(var.address_space) == "255.255.0.0"
      error_message = "The IP Range must be /16"
    }
    postcondition {
      condition = self.location  == "westeurope"
      error_message = "Location must be westeurope"
    }
  }
}
```

3. Finally, run the Terraform workflow and check that no Terraform error is displayed in the console output.

How it works...

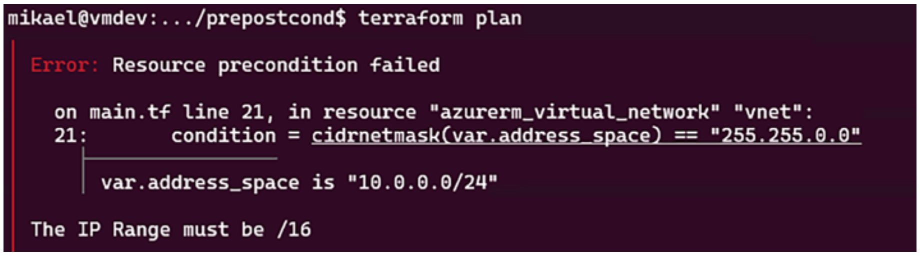
In *Step 1*, we added the first custom check, which corresponds to the precondition that will be run just before the plan. The `precondition` block is new inside the `lifecycle` block metadata. Let's see this precondition in detail:

```
precondition {
      condition = cidrnetmask(var.address_space) == "255.255.0.0"
      error_message = "The IP Range must be /16"
}
```

The `precondition` block contains two properties:

1. The `condition`, that is, the code for the check. Here, we check that the `cidrmask` of the value of the `address_mask` variable is equal to "`255.255.0.0`", that is, that the IP range is `/16`.
2. The `error_message`, that is, the error message that is displayed in the console output if the check returns `false`.

To test this precondition, if we set the value of the `address_space` variable to `"10.0.0.0/24"`, the `terraform plan` execution returns this output:



```
mikael@vmdev:.../prepostcond$ terraform plan

Error: Resource precondition failed

  on main.tf line 21, in resource "azurerm_virtual_network" "vnet":
  21:         condition = cidrnetmask(var.address_space) == "255.255.0.0"
     ├────────────────
     │ var.address_space is "10.0.0.0/24"

The IP Range must be /16
```

Figure 2.18: Precondition custom validation error

The error message is displayed and the `terraform plan` command doesn't continue.

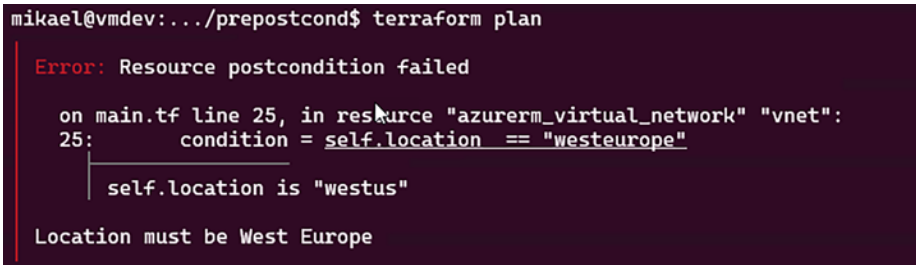
Then in *Step 2*, we add the check for testing the region (the data center's location) of the Azure virtual network, which must be equal to "`westeurope`". To do this, we add a `postcondition` block inside the `lifecycle` metadata with the following configuration:

```
postcondition {
      condition = self.location  == "westeurope"
      error_message = "Location must be West Europe"
}
```

In the configuration above, we set the `condition` property by using the `self` keyword to refer to the current resource (in this case this is the Azure virtual network) and we set the error message.

Note that the `self` keyword can be used only on postconditions, at the moment that all properties are determined, which is only after the `terraform plan` command has been run.

To test this `postcondition`, we set the location to `"westus"` and we get this `terraform plan` output:



```
mikael@vmdev:.../prepostcond$ terraform plan

Error: Resource postcondition failed

  on main.tf line 25, in resource "azurerm_virtual_network" "vnet":
  25:         condition = self.location  == "westeurope"
     ├────────────────
     │ self.location is "westus"

Location must be West Europe
```

Figure 2.19: Postcondition custom validation error

We can see that the error message is displayed.

See also

- The documentation of the pre- and postconditions is available at `https://www.terraform.io/language/meta-arguments/lifecycle#custom-condition-checks`.
- Here's a tutorial on pre- and postconditions on modules: `https://learn.hashicorp.com/tutorials/terraform/custom-conditions`.
- Here's a blog post about pre and postconditions: `https://spacelift.io/blog/terraform-precondition-postcondition`.
- Ned Bellavance's video on pre and postconditions is available here: `https://www.youtube.com/watch?v=55ZLu8tSnvk`.

Using checks for infrastructure validation

In the previous recipe, *Adding custom pre and postconditions*, we learned that it is possible to add pre- or postcondition validation inside the resource configuration.

In Terraform version 1.5 and newer, it's possible to add infrastructure validation directly in the Terraform configuration, which allows us to check that the provisioned infrastructure is working as intended.

Let's get started!

Getting ready

In this recipe, we will provision a new Azure App Service instance using a Terraform configuration and inside this same Terraform configuration, we will check that the provisioned App Service instance is running and returns an HTTP Status code equal to 200.

> Note that in this recipe we will not go into detail about the Terraform configuration for the Azure App Service. We will only look directly at the availability check of the App Service.

So for this recipe, we will start with the Terraform configuration available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/sample-app`, which we will copy into another folder called `check`.

In the recipe we will learn how to check the App Service instance's availability. The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP02/check`.

How to do it...

To check the provisioned infrastructure, perform the following steps:

1. In the `main.tf` file that is copied into the `check` folder, add the following Terraform configuration:

```
check "response" {
  data "http" "webapp" {
    url      = "https://${azurerm_linux_web_app.app.default_
hostname}"
insecure=True
  }

  assert {
    condition     = data.http.webapp.status_code == 200
    error_message = "Web app response is ${data.http.webapp.status_
code}"
  }
}
```

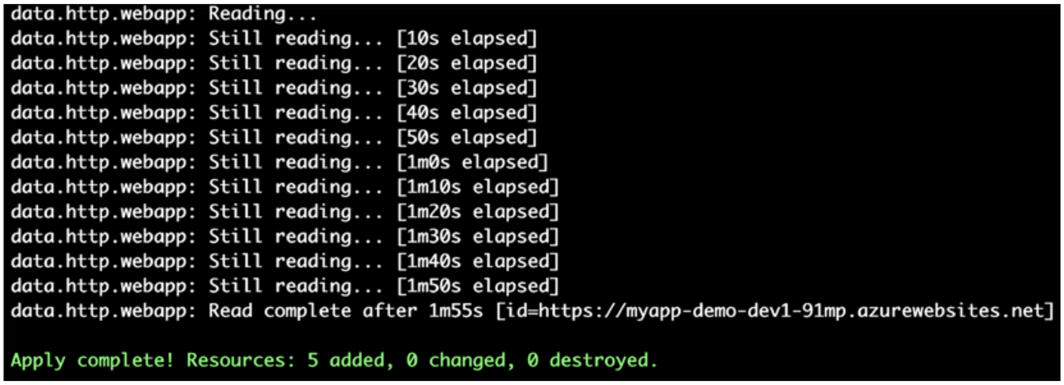
2. In this folder, execute the basic Terraform workflow by running the `terraform init`, `plan`, and `apply` commands.

How it works...

In *Step 1*, we added the `check` block, which contains:

- A `data` HTTP source that performs an HTTP GET on the given URL. Here, we use the default hostname of the web app in the URL property. For more details about the data HTTP source block, refer to the documentation here: `https://registry.terraform.io/providers/hashicorp/http/latest/docs/data-sources/http`.
- An `assert` block that evaluates the response of the `data` source by checking that the HTTP code is equal to `200` (status code ok). If this evaluation returns `false`, then the `assert` block displays the configured `error_message`.

In *Step 2*, we run the Terraform workflow to create the Azure web app and check its availability. The following image shows the output of the `terraform apply` command:

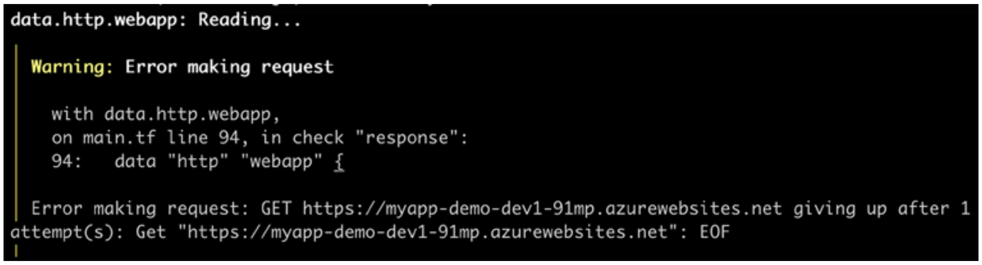


```
data.http.webapp: Reading...
data.http.webapp: Still reading... [10s elapsed]
data.http.webapp: Still reading... [20s elapsed]
data.http.webapp: Still reading... [30s elapsed]
data.http.webapp: Still reading... [40s elapsed]
data.http.webapp: Still reading... [50s elapsed]
data.http.webapp: Still reading... [1m0s elapsed]
data.http.webapp: Still reading... [1m10s elapsed]
data.http.webapp: Still reading... [1m20s elapsed]
data.http.webapp: Still reading... [1m30s elapsed]
data.http.webapp: Still reading... [1m40s elapsed]
data.http.webapp: Still reading... [1m50s elapsed]
data.http.webapp: Read complete after 1m55s [id=https://myapp-demo-dev1-91mp.azurewebsites.net]

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

Figure 2.20: Check infrastructure validation is successful

There's more...

- Unlike the pre and postconditions, checking with the `check` block does not block resource provisioning if the assertion returns `false`. Instead, just a warning message in the output is displayed as shown in the following screenshot:



```
data.http.webapp: Reading...
│
│ Warning: Error making request
│
│   with data.http.webapp,
│   on main.tf line 94, in check "response":
│   94:   data "http" "webapp" {
│
│ Error making request: GET https://myapp-demo-dev1-91mp.azurewebsites.net giving up after 1
attempt(s): Get "https://myapp-demo-dev1-91mp.azurewebsites.net": EOF
│
```

Figure 2.21: Check infrastructure validation on error with warning message

- Additionally, in this recipe, we demonstrated a check sample using a `data` source. However, this `data` source isn't mandatory and its use will depend on the specific requirements of your infrastructure checks. For more details about the `check` block, read the tutorial at `https://developer.hashicorp.com/terraform/tutorials/configuration-language/checks`.

See also

- The `check` block documentation is available at `https://developer.hashicorp.com/terraform/language/checks`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

3

Scaling Your Infrastructure with Terraform

In the previous chapter, we learned the basics of how to use Terraform language primitives to provision infrastructure efficiently. One of the advantages of **Infrastructure as Code** (**IaC**) is that it allows you to provision infrastructure on a large scale much faster than manual provisioning.

When writing IaC, it is also important to apply the development and clean code principles that developers have formulated over the years.

One of these principles is **Don't Repeat Yourself** (**DRY**), which means not duplicating code. The following link explains the DRY principles in detail: `https://thevaluable.dev/dry-principle-cost-benefit-example/`.

In this chapter, we will learn how to use expressions from the Terraform language, such as `count`, `maps`, `collections`, and `array`, and the use of the `dynamic` block.

We will learn that these expressions allow us to write simple Terraform configurations to deploy infrastructure in multiple environments and provide it with multiple resources, without having to duplicate code.

In this chapter, we will cover the following recipes:

- Provisioning infrastructure in multiple environments
- Provisioning multiple resources with the `count` meta-argument
- Using maps
- Looping over a map of objects

- Generating multiple blocks with the dynamic block
- Filtering maps

Technical requirements

This chapter does not have any technical prerequisites. However, it is advisable to have already read the previous chapter.

The source code of this chapter is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP03/`.

Check out the following video to see the code in action: `https://bit.ly/2R5GSBN`.

Provisioning infrastructure in multiple environments

In the same way that we deploy an application to several environments (development, QA, and production), we also need to provision infrastructure in these different environments to support these applications.

The question that often arises is how to write a maintainable and scalable Terraform configuration that would allow us to provision infrastructure for multiple environments.

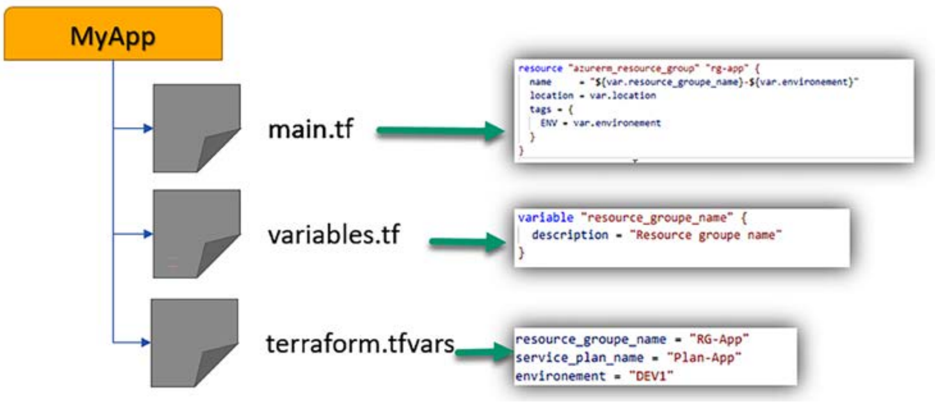
To answer this question, it is important to know that there are several solutions for organizing Terraform configuration topologies that will allow this provisioning.

In this recipe, we will look at two Terraform configuration structure hierarchies that will allow us to deploy an Azure infrastructure to multiple environments.

Getting ready

To fully understand this recipe, you will need to have a good understanding of the notion of variables, as discussed in *Chapter 2*, *Writing Terraform Configurations*, in the recipe *Manipulating variables*.

The goal of the Terraform configuration that we are going to write is to deploy an Azure app service for a single environment. Its code is distributed in the following files:



```
resource "azurerm_resource_group" "rg-app" {
  name     = "${var.resource_groupe_name}-${var.environement}"
  location = var.location
  tags = {
    ENV = var.environement
  }
}
```

```
variable "resource_groupe_name" {
  description = "Resource groupe name"
}
```

```
resource_groupe_name = "RG-App"
service_plan_name = "Plan-App"
environement = "DEV1"
```

Figure 3.1: Terraform configuration structure

In the diagram above, we can see the following:

- The `main.tf` file contains the Terraform configuration of the resources to be provisioned.
- The `variables.tf` file contains the declaration of the variables.
- The `terraform.tfvars` file contains the values of the variables.

The Terraform source code for this basic example is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP03/myApp/simple-env`.

What is important in this recipe is not the content of the configuration, but the folder structure and the Terraform commands to be executed.

How to do it...

Follow these steps to implement the first Terraform configuration folder hierarchy:

1. In an empty folder, create a separate directory per environment: one for `dev`, one for `QA`, and one for `production`.
2. Copy the base Terraform configuration into each of these directories identically.
3. Then, in each of these directories, modify the values of the `terraform.tfvars` file with the information that is specific to the environment. Here is an extract of each of these `terraform.tfvars` files:

```
resource_group_name = "RG-Appdemo"
service_plan_name = "Plan-App"
environment = "DEV" #name of the environment to change
```

4. Finally, to provision each of these environments, inside each of these directories, execute the basic Terraform execution workflow by running the `terraform init`, `terraform plan -out="out.tfplan"`, and `terraform apply out.tfplan` commands.

Follow these steps to implement the second hierarchy of the Terraform configuration folder:

1. In the folder that contains our basic Terraform configuration, create three subdirectories: `dev`, `test`, and `production`.
2. Then, in each of these subdirectories, copy only the `terraform.tfvars` base file, in which we modify the variables with the correct values of the target environments. The following is an extract from each `terraform.tfvars` file:

```
resource_group_name = "RG-Appdemo"
service_plan_name = "Plan-App"
environment = "DEV" #name of the environment to change
```

3. Finally, to provision each of these environments, go to the folder of the Terraform configuration and execute the following commands:

```
terraform init
terraform plan -out="out.tfplan"
terraform apply out.tfplan
terraform.tfvars"
```

How it works...

In the first topology, we duplicate the same Terraform configuration for each environment and just change the values of the variables in the `terraform.tfvars` file of each folder.

By doing this, we get the following folder structure:

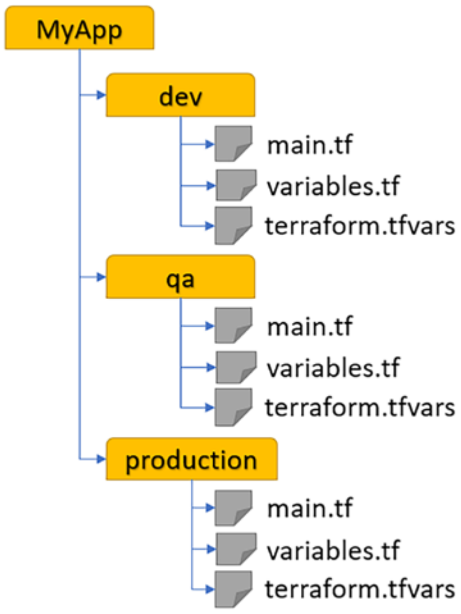


Figure 3.2: Terraform configuration structure by environment

Terraform is then executed with the basic Terraform commands. This structure can be used if the infrastructure does not contain the same resources for each environment. This is because duplicating all the Terraform configurations in each environment folder offers us the advantage of being able to easily add or remove resources for one environment without affecting the other environments.

However, this is duplicate code, which implies that this code must be maintained several times (we must modify the infrastructure for all environments, make changes to the Terraform configuration, and so on).

In the second topology, we keep the Terraform configuration in the common base for all environments and have just one `terraform.tfvars` file per environment, named after the name of the environment. By doing this, we get the following folder structure:

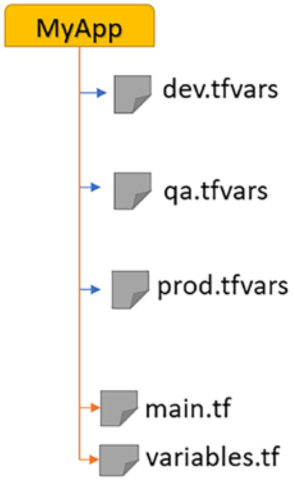


Figure 3.3: Terraform configuration structure with environment configuration

As for the execution of the Terraform configuration, we have added the `-var-file` option to the `plan` and `apply` commands. This structure can be used if the infrastructure is the same for all environments but only the configuration changes.

The advantage of this hierarchy is that we have only one common piece of Terraform resource code (in the `main.tf` and `variables.tf` files), and just one `terraform.tfvars` file to fill in, so we will have to make a few changes in case of code evolution or a new environment.

On the other hand, the changes that were made to the Terraform `main.tf` configuration will apply to all the environments, which in this case requires more testing and verification.

Be careful though; the simple use of these `tfvars` files as described in this recipe implies that no matter the environment, the Terraform State file will be the same, which may impact your provisioned infrastructure.

It is, therefore, necessary to add the configuration of the Terraform State backend with a file `<env>-backend.tfvars` (which will be used as an argument to the `terraform init` command) to this file structure. For more information, read the documentation here: `https://developer.hashicorp.com/terraform/cli/commands/init#backend-initialization`.

See also

- There are other solutions to Terraform configuration folder structure topologies, as we will discuss in *Chapter 7, Sharing Terraform Configuration with Modules*.
- Documentation regarding the `-var-file` option of the `plan` and `apply` commands is available at `https://www.terraform.io/docs/commands/plan.html`.
- An article explaining the best practices surrounding Terraform configuration can be found at `https://www.terraform-best-practices.com/code-structure`.
- The following blog post explains the folder structure for Terraform configuration for production: `https://www.hashicorp.com/blog/structuring-hashicorp-terraform-configuration-for-production`.

Provisioning multiple resources with the count meta-argument

In many corporate scenarios, there is a need to provide infrastructure and to consider the so-called horizontal scalability, that is, *N* identical resources that will reduce the load on individual resources (such as compute instances) and the application.

The challenges we will have to face are as follows:

- Writing Terraform configuration that does not require duplicate code for each instance of the identical resources to be provisioned
- Being able to rapidly increase or reduce the number of instances of these resources

We will see in this recipe how Terraform makes it possible to provision N instances of resources quickly and without the duplication of code.

Getting ready

To begin, we will use a Terraform configuration that allows us to provision one Azure app service, which is in a `main.tf` file and of which the following is an extract:

```
resource "azurerm_linux_web_app" "app" {
  name                = "${var.app_name}-${var.environment}"
  location            = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  app_service_plan_id = azurerm_app_service_plan.plan-app.id
}
```

The purpose of this recipe is to apply and modify this Terraform configuration to provision *N* Azure App Service instances identical to the one already described in the base code, with just a slight difference in the names, which use an incremental index number starting at 1.

The source code of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP03/count`.

How to do it...

To create multiple identical resources, perform the following steps:

1. In the `variables.tf` file, we add the following variable:

   ```
   variable "webapp_count" {
       description = "Number of App to create"
   }
   ```

2. In the `terraform.tfvars` file, we give a value for this new variable as follows:

   ```
   webapp_count = 2
   ```

3. In the `main.tf` file, we modify the resource code of `azurerm_linux_web_app` in the following way:

   ```
   resource "azurerm_linux_web_app" "app" {
     count = var.webapp_count
     name = "${var.app_name}-${var.environment}-${random_string.random.
   result}-${count.index+1}"
     location = azurerm_resource_group.rg.location
     resource_group_name = azurerm_resource_group.rg.name
     service_plan_id = azurerm_service_plan.plan.id
   …
   }
   ```

4. (Optionally) in a new `outputs.tf` file, we add the output values with the following code:

   ```
   output "app_service_names"{
     value = azurerm_linux_web_app.app.*.name
   }
   ```

How it works...

In *Step 1*, we add an `nb_webapp` variable, which will contain the number of Azure App Service instances to write, which we then instantiate in *Step 2* in the `terraform.tfvars` file.

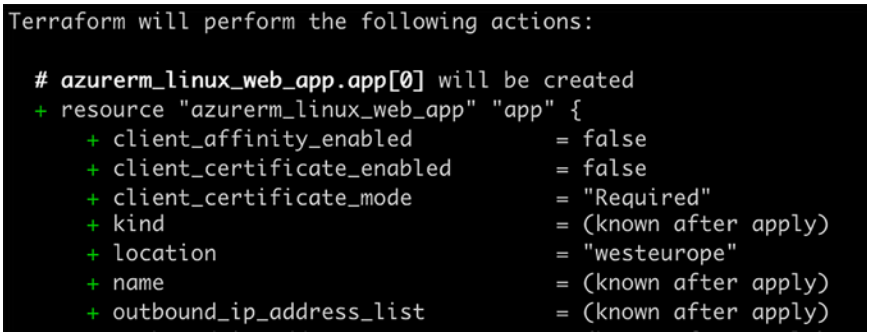
Then in *Step 3*, in the `azurerm_linux_web_app` resource, we add the Terraform `count` meta-argument (which is available for all `resources` and `data` Terraform blocks) , which takes the `nb_webapp` variable created previously as a value.

Moreover, in the `name` of the `azurerm_linux_web_app` resource, we add the suffix with the current index of the `count` that we increment by 1 (starting from 1, and not from 0, i.e., we reflect the fact that `count` indexes start from 0), using the Terraform instruction `count.index + 1`.

Finally, and optionally, in *Step 4*, we can add an output that will contain the names of the App Service instances that have been provisioned.

When executing the `terraform plan` command of this recipe with the `nb_webapp` variable equal to `2`, we can see that the two App Service instances have been provisioned.

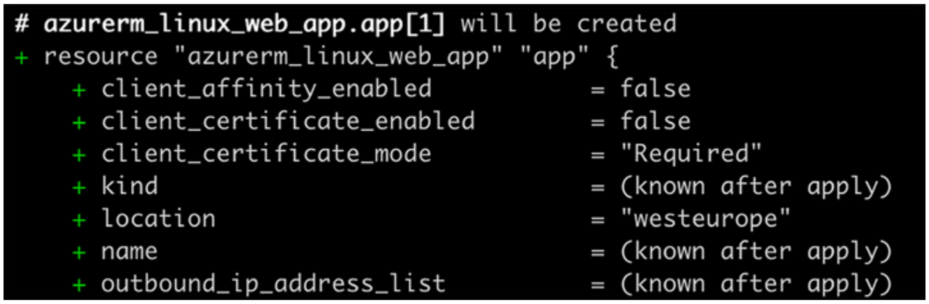
The following screenshots show an extract of this `terraform plan` command, with the first image displaying the preview changes for the first Azure App Service:



```
Terraform will perform the following actions:

  # azurerm_linux_web_app.app[0] will be created
  + resource "azurerm_linux_web_app" "app" {
      + client_affinity_enabled           = false
      + client_certificate_enabled        = false
      + client_certificate_mode           = "Required"
      + kind                              = (known after apply)
      + location                          = "westeurope"
      + name                              = (known after apply)
      + outbound_ip_address_list          = (known after apply)
```

Figure 3.4: Terraform count index

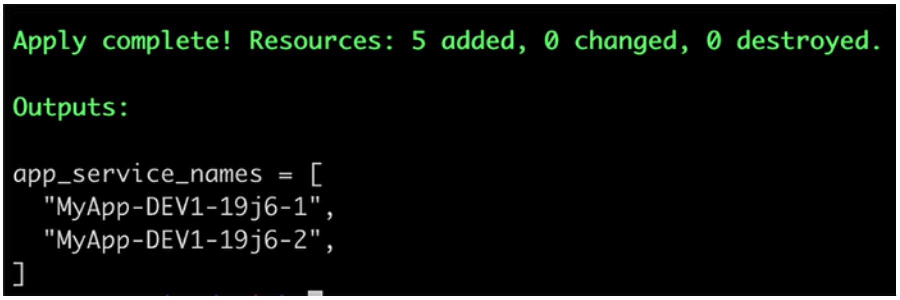
The following screenshot, which is the continuation of the `terraform plan` command, displays the preview changes of the second App Service instance:



```
# azurerm_linux_web_app.app[1] will be created
+ resource "azurerm_linux_web_app" "app" {
    + client_affinity_enabled           = false
    + client_certificate_enabled        = false
    + client_certificate_mode           = "Required"
    + kind                              = (known after apply)
    + location                          = "westeurope"
    + name                              = (known after apply)
    + outbound_ip_address_list          = (known after apply)
```

Figure 3.5: Second Terraform count index

And when the changes are applied, the output is displayed:



```
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

app_service_names = [
  "MyApp-DEV1-19j6-1",
  "MyApp-DEV1-19j6-2",
]
```

Figure 3.6: Terraform output using count

As you can see in the output, we have a list containing the names of the two generated App Service instances.

There's more...

As we discussed in *Chapter 2*, *Writing Terraform Configurations*, in the recipe *Manipulating variables*, we can also use the `-var` option of the `terraform plan` and `apply` commands to very easily increase or decrease the number of instances of this resource, without having to modify the Terraform configuration.

In our case, for example, we could use the following `plan` and `apply` commands:

```
terraform plan -var "webapp_count=5" -out out.tfplan
terraform apply out.tfplan
```

However, with this option, we lose one of the major benefits of IaC (where everything is written in code, and thus, you have a history of the changes made to the infrastructure).

Moreover, it should be noted that lowering the `webapp_count` value removes the last resources from the index, and it is not possible to remove resources that are in the middle of the index – therefore, it is best to keep all resource instances with count the same (e.g., don't use conditional logic within other attributes based on `count.index`). This can be addressed using the `for_each` expression, which we will see in the *Looping over a map of objects* recipe in this chapter.

In addition, thanks to the count meta-argument we have just seen and the condition expressions we studied in the *Writing conditional operations* recipe of *Chapter 2*, *Writing Terraform Configurations*, we can make the provisioning of resources optional in a dynamic way, as shown in the following code snippet:

```
resource "azurerm_application_insights" "appinsight_app" {
  count = var.use_appinsight == true ? 1 : 0
  ....
}
```

In this code, we have indicated to Terraform that if the `use_appinsight` variable is `true`, then the `count` meta-argument value is `1`, which will allow us to provision one Azure Application Insights resource. In the opposite case, where the `use_appinsight` variable is `false`, the `count` meta-argument value is `0`, and in this case, Terraform does not provision an Application Insights resource instance.

And so, Terraform configuration can be used generically for all environments or all applications and make their provisioning dynamic and conditional, according to variables.

This technique, also called **feature flags**, is applied in the development world, but we see here that we can also apply it to IaC by using this technique to provision *N* identical instances without duplicate code.

As we have seen in this recipe, the `count` meta-argument allows you to quickly provision several resources that are identical in their characteristics.

We will study, in the *Looping over object collections* recipe of this chapter, how to provision several resources of the same nature, but with different properties.

See also

For more information on the `count` meta-argument, refer to the documentation at `https://www.terraform.io/language/meta-arguments/count`.

Using maps

So far in this book, we have studied sample code using standard variable types (string, numeric, or Boolean). However, the Terraform language has other types of variables such as lists, sets, maps, tuples, and even more complex object variables.

Among these variable types are maps, which are represented by a collection of key-value elements and are widely used to write dynamic and scalable Terraform configurations.

Maps can have several uses, which are as follows:

- To put all the properties of a block in a Terraform resource into a single variable
- To have a key-value reference table of elements that will be used in the Terraform configuration
- To declare data for multiple instances of the same resource, which will then iterate over the map via `for_each`

In this recipe, we will see a simple and practical case of using a `map` variable. For this recipe, the role of `map` is to dynamically define all the tags of an Azure resource, but it can be applied to any Terraform resource provider.

Getting ready

For this recipe, we start with a basic Terraform configuration that allows us to provision a resource group and an App Service instance in Azure.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP03/map`.

In this recipe, we will illustrate the use of maps in two use cases, which are as follows:

- The implementation of the tags of this resource group
- The app settings properties of App Service

How to do it…

Perform the following steps:

1. In the `variables.tf` file, we add the following variable declarations:

   ```
   variable "tags" {
     type = map(string)
     description = "Tags"
     default = {}
   }

   variable "app_settings" {
     type = map(string)
     description = "App settings of the Web App"
     default = {}
   }
   ```

2. Then, in the `terraform.tfvars` file, we add this code:

```
tags = {
  ENV = "DEV1"
  CODE_PROJECT = "DEMO"
}

app_settings = {
  KEY1 = "VAL1"
}
```

3. Finally, we modify the `main.tf` file with the following code:

```
resource "azurerm_resource_group" "rg-app" {
  name = "${var.resource_group_name}-${var.environement}-${random_
string.random.result}"
  location = var.location
  tags = var.tags
}

resource "azurerm_linux_web_app" "app" {
  name = "${var.app_name}-${var.environment}-${random_string.random.
result}"
  location = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id = azurerm_service_plan.plan-app.id

  site_config {}
}
```

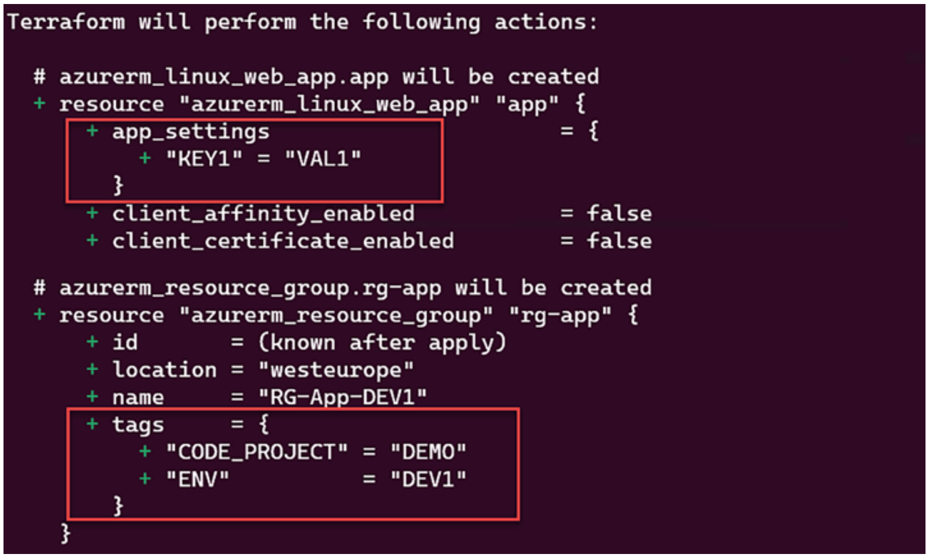
How it works...

In *Step 1*, we have declared two variables for which we have specified their type, which is `map (string)`. It will be composed of a key, which has a value of type *string*. Moreover, given that these variables can be omitted and their values are, therefore, optional, we have assigned them an empty default value, which is `{}` for `map`.

Then, in *Step 2*, we defined the values of these two variables with the tags for the resources, as well as the `app_settings` for App Service.

Finally, in *Step 3*, we use these variables in the Terraform configuration that provides the Resource Group and the App Service.

The following screenshot shows a sample (extract) of the execution of the `terraform plan` command in this recipe:



```
Terraform will perform the following actions:

  # azurerm_linux_web_app.app will be created
  + resource "azurerm_linux_web_app" "app" {
      + app_settings                          = {
          + "KEY1" = "VAL1"
        }
      + client_affinity_enabled               = false
      + client_certificate_enabled            = false

  # azurerm_resource_group.rg-app will be created
  + resource "azurerm_resource_group" "rg-app" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "RG-App-DEV1"
      + tags     = {
          + "CODE_PROJECT" = "DEMO"
          + "ENV"          = "DEV1"
        }
    }
```

Figure 3.7: Terraform key-value usage

We can see in the previous screenshot that the `app_settings` and `tags` properties are populated with the values of the `map` variables.

There's more...

To go further, we can see that it is also possible to merge maps; that is, to merge two maps, we can use the `merge` function, which is native to Terraform.

The following steps show how to use this function to merge the app settings properties of App Service:

1. In the `variables.tf` file, we create a `custom_app_settings` variable that will contain the custom app settings provided by the user:

```
variable "custom_app_settings" {
  description = "Custom app settings"
  type = map(string)
  default = {}
}
```

2. In the `terraform.tfvars` file, we instantiate this variable with a custom map:

```
custom_app_settings = {
  APP = "1"
}
```

3. Finally, in the `main.tf` file, we use a local variable to define the default app settings, and in the `azurerm_linux_web_app` resource, we use the `merge` function to merge the default app settings with the custom app settings:

```
resource "azurerm_linux_web_app" "app" {
  name = "${var.app_name}-${var.environement}"
  location = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id = azurerm_service_plan.plan-app.id

  site_config {}
}
```

In the preceding code, we have defined default app settings properties for Azure App Service, and the user can enrich these settings if needed by adding custom app settings.

In this recipe, we studied the use of maps. But if we want to use more complex data structures with values of different types, then we will use object variables, as explained in the documentation at `https://www.terraform.io/language/expressions/type-constraints#complex-types`.

In the following recipe, we will discuss how to iterate on the list of key-value elements that constitute a `map` variable.

See also

The documentation relating to the `merge` function is available at `https://www.terraform.io/docs/configuration/functions/merge.html`.

Looping over a map of objects

We have seen in the previous recipes of the chapter the use of the `count` property, which allows us to provision *N* identical resources, as well as the use of map variables, which allow values to be of type `object`.

In this recipe, we will discuss how to provision *N* resources of the same type but with different properties, using the loop functionalities included in Terraform since version 0.12.

Getting ready

We'll start with a basic Terraform configuration that allows you to deploy a single App Service in Azure.

The basic Terraform configuration is as follows:

```
resource "azurerm_app_service" "app" {
  name = "${var.app_name}-${var.environement}"
  location = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  app_service_plan_id = azurerm_app_service_plan.plan-app.id
}
```

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP03/list_map`.

How to do it…

Perform the following steps:

1. In the `variables.tf` file, we add the following Terraform configuration:

```
variable "web_apps" {
  description = "List of App Service to create"
  type = map(object({
  name = string
  location = optional(string, "westeurope")
  serverdatabase_name = string
}))
}
```

2. In the `terraform.tfvars` file, we add the following configuration:

```
web_apps = {
  webapp1 = {
   "name" = "webappdemobook1"
   "location" = "westeurope"
   "serverdatabase_name" = "server1"
  },

  webapp2 = {
    "name" = "webapptestbook2"
    "serverdatabase_name" = "server2"
  }
}
```

3. In the `main.tf` file, we modify the code of App Service with the following configuration:

```
resource "azurerm_linux_web_app" "app" {
  for_each = var.web_apps
  name = each.value["name"]
  location = lookup(each.value, "location", "westeurope")
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id = azurerm_service_plan.plan-app.id


  connection_string {
    name = "DataBase"
    type = "SQLServer"
    value = "Server=${each.value["serverdatabase_name"]};Integrated
Security=SSPI"
  }
}
```

4. Finally, in the `outputs.tf` file, we add the following code:

```
output "app_service_names" {
  value = [for app in azurerm_linux_web_app.app : app.name]
}
```

How it works...

In *Step 1*, we declared a new variable that was a map of objects and contained the details of each `map` property, including the following:

- The `name` is type string.
- The `location` is optional, of type string, and of any value, provided the default value is `westeurope`.
- The `serverdatabase_name` is type string.

In *Step 2*, we instantiate this variable with a map of objects that will be the properties of each app service. In this list, we have two App Service instances in which we specify the properties in the form of a map with the name, the version of the framework, the Azure region location, and the name of the database server of the application that will be used in the app service.

In *Step 3*, in the `azurerm_linux_web_app` resource, we use the `for_each` expression, which allows us to loop on maps.

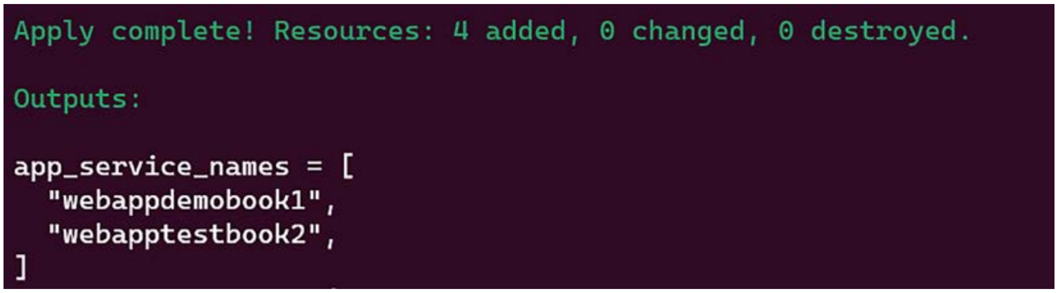
Then, for each property of the `azurerm_linux_web_app` resource, we can use the short expression `each.value["<property name>"]`, or the `lookup` function integrated into Terraform that takes in the following parameters:

- The current element of the `for_each` expression with `each.value`, thus the line of the list
- The name of the property of the map, which in our sample is `location`

Then comes the third parameter of this `lookup` function, which is not mandatory. It allows you to specify the value to use if the property is not present in the map. In the `location` property, we used the Azure West Europe region (`westeurope`) as the default value.

Finally, in *Step 4*, we created an output that uses the `for` expression to iterate on the list of resources that have been provisioned and export their names as output.

The result of this output is shown in the following screenshot:



```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

app_service_names = [
  "webappdemobook1",
  "webapptestbook2",
]
```

Figure 3.8: Terraform output of the for expression

In the previous screenshot, we can see the result of the output, which displays the name of the two provisioned Azure App Service instances in the console.

There's more...

In this recipe, we have learned expressions and functions from the Terraform language that allow us to provision resource collections.

I advise you to take a good look at the articles and documentation on the `for` expression and `for_each` meta-argument. The `lookup` and `element` functions can be handy. However, whenever possible, it is recommended to use the native syntax (such as `var_name[42]` and `var_map["key"]`) to access elements of a map, list, or set.

It is obvious that in this recipe, we have used simple resources such as Azure App Service, but these methods can also be applied to more property-rich resources such as virtual machines.

See also

- Documentation on loops with `for` and `for_each` is available at `https://www.terraform.io/language/meta-arguments/for_each`, and the article on these loops can be found at `https://www.hashicorp.com/blog/hashicorp-terraform-0-12-preview-for-and-for-each/`.
- Read this documentation to see the difference between the `count` and `for_each` meta-arguments: `https://www.terraform.io/language/meta-arguments/count#when-to-use-for_each-instead-of-count`.
- The documentation on the `lookup` function is available at `https://www.terraform.io/docs/configuration/functions/lookup.html`.

Generating multiple blocks with the dynamic block

Terraform resources are defined by the following elements:

- Properties that are in the form `property name = value`, which we have seen several times in this book
- Blocks that represent a grouping of properties, such as the `site_config` block inside the `azurerm_linux_web_app` resource

Depending on the Terraform resource, a block can be present once or even multiple times in the same resource, such as the `security_rule` block inside the `azurerm_network_security_group` resource (see the documentation, for example, at `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/network_security_group`).

One of the great features of Terraform is the `dynamic` block, which allows us to loop the blocks in resources.

In this recipe, we will see how to use the `dynamic` block to provision an `azurerm_network_security_group` resource in Azure, which contains a list of security rules.

Getting ready

To get started, we don't need any basic Terraform configuration. In this recipe, we will use a Terraform file that allows us to create an Azure resource group, in which we will create a network security group.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP03/dynamics`.

How to do it...

To use the `dynamic` block, perform the following steps:

1. In the `variables.tf` file, we add the following code:

```
variable "nsg_rules" {
 description = "List of NSG rules"
 type        = list(object({
    name                       = string
    priority                   = number
    direction                  = string
    access                     = string
    protocol                   = string
    source_port_range          = string
    destination_port_range     = string
    source_address_prefix      = string
    destination_address_prefix = string
  }))
}
```

2. In the `terraform.tfvars` file, we add the following code:

```
ngs_rules = [
{
 name = "rule1"
 priority = 100
 direction = "Inbound"
 access = "Allow"
 protocol = "Tcp"
 source_port_range = "*"
 destination_port_range = "80"
 source_address_prefix = "*"
 destination_address_prefix = "*"
},
{
 name = "rule"
 priority = 110
 direction = "Inbound"
 access = "Allow"
 protocol = "Tcp"
 source_port_range = "*"
 destination_port_range = "22"
 source_address_prefix = "*"
 destination_address_prefix = "*"
}
]
```

3. In the `main.tf` file, we add the code for the network security group with the following code:

```
resource "azurerm_network_security_group" "example" {
  name = "acceptanceTestSecurityGroup1"
  location = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name

  dynamic "security_rule" {
     for_each = var.ngs_rules
     content {
      name = security_rule.value["name"]
      priority = security_rule.value["priority"]
      direction = security_rule.value["direction"]
      access = security_rule.value["access"]
      protocol = security_rule.value["protocol"]
      source_port_range = security_rule.value["source_port_range"]
      destination_port_range = security_rule.value["destination_
port_range"]
      source_address_prefix = security_rule.value["source_address_
prefix"]
      destination_address_prefix = security_rule.value["destination_
address_prefix"]
     }
  }
}
```

How it works...

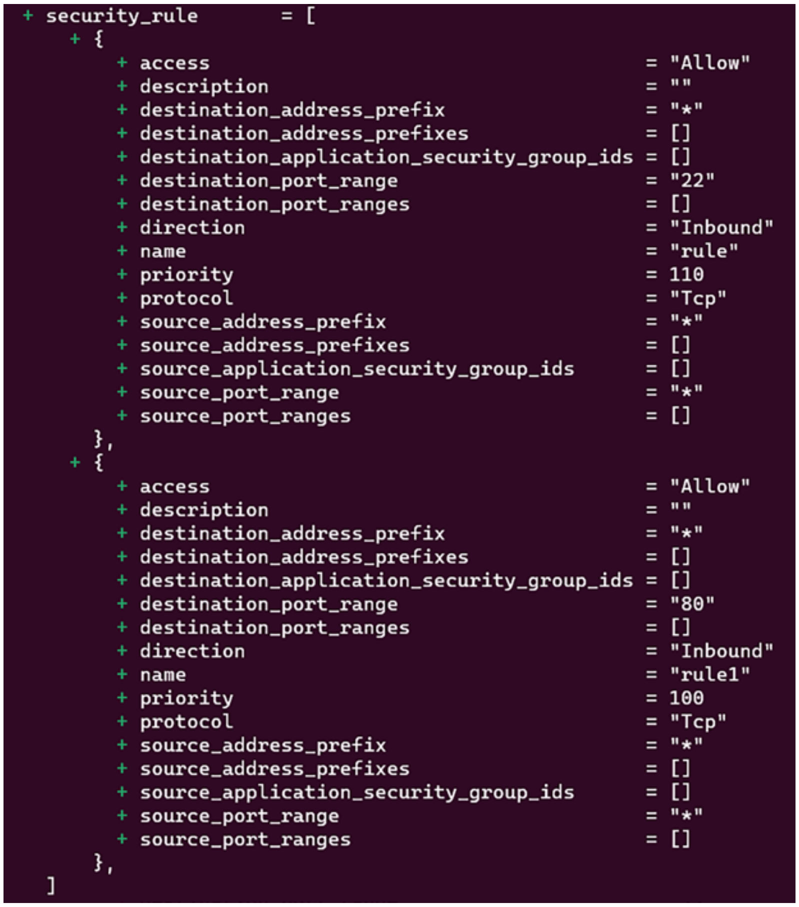
In *Step 1*, we create an `nsg_rules` variable of type `any`, which will contain the list of rules in the `map` format.

Then, in *Step 2*, we instantiate this `nsg_rules` variable with the list of rules and their properties.

Finally, in *Step 3*, in the `azurerm_network_security_group` resource, we add the `dynamic` instruction, which allows us to generate *N* blocks of `security_rule`.

In this `dynamic` Terraform block, we make a `for_each` loop (as seen in the *Looping over a map of objects* recipe earlier in this chapter), which will iterate over the elements of the `nsg_rules` variable and map each property of the resource to the maps of the list.

The following screenshot shows the execution of the `terraform plan` command:



```
+ security_rule       = [
    + {
        + access                                     = "Allow"
        + description                                = ""
        + destination_address_prefix                 = "*"
        + destination_address_prefixes               = []
        + destination_application_security_group_ids = []
        + destination_port_range                     = "22"
        + destination_port_ranges                    = []
        + direction                                  = "Inbound"
        + name                                       = "rule"
        + priority                                   = 110
        + protocol                                   = "Tcp"
        + source_address_prefix                      = "*"
        + source_address_prefixes                    = []
        + source_application_security_group_ids      = []
        + source_port_range                          = "*"
        + source_port_ranges                         = []
      },
    + {
        + access                                     = "Allow"
        + description                                = ""
        + destination_address_prefix                 = "*"
        + destination_address_prefixes               = []
        + destination_application_security_group_ids = []
        + destination_port_range                     = "80"
        + destination_port_ranges                    = []
        + direction                                  = "Inbound"
        + name                                       = "rule1"
        + priority                                   = 100
        + protocol                                   = "Tcp"
        + source_address_prefix                      = "*"
        + source_address_prefixes                    = []
        + source_application_security_group_ids      = []
        + source_port_range                          = "*"
        + source_port_ranges                         = []
      },
  ]
```

Figure 3.9: Terraform dynamic block

We can see the list of security rules in the preceding output.

There's more...

If you want to render the presence of a block conditionally, you can also use the conditions in the `dynamic` expression, as shown in the following code sample:

```
resource "azurerm_linux_virtual_machine" "virtual_machine" {
...
  dynamic "boot_diagnostics" {
     for_each = local.use_boot_diagnostics == true ? [1] : []
     content {
     storage_account_uri = "https://storageboot.blob.core.windows.net/"
     }
  }
}
```

In this example, in the `for_each` expression of the `dynamic` expression, we have a conditional expression that returns a list with one element if the local value, `use_boot_diagnostics`, is `true`. Otherwise, this condition returns an empty list that will not make the `boot_diagnostics` block appear in the `azurerm_linux_virtual_machine` resource.

See also

- Documentation on `dynamic` expressions is available at `https://www.terraform.io/language/expressions/dynamic-blocks`.
- Another example of a Terraform guide on `dynamic` expressions is available at `https://github.com/hashicorp/terraform-guides/tree/master/infrastructure-as-code/terraform-0.12-examples/advanced-dynamic-blocks`.

Filtering maps

In the previous recipes of this chapter, we learned different use cases for looping over collections using the `for` and `for_each` expressions.

In some scenarios, we may need to create resources from an existing list, but the requirement is to filter only some elements from this list.

In this recipe, we will learn how to filter a list to provision resources.

Let's get started!

Getting ready

To complete this recipe, you'll need to know about looping with the `for_each` expression, which was covered in the *Looping over object collections* recipe of this chapter.

The starting element of this recipe is the following list of objects. This object list contains some properties of the resources needed for our App Service use cases:

```
web_apps = [
  webapp1 = {
    "name" = "webapptestbook1"
    "os"   = "Linux"
  },
  webapp2 = {
    "name" = "webapptestbook2"
    "os"   = "Linux"
  },
  webapp3 = {
    "name" = "webapptestbook3"
    "os"   = "Windows"
  }
]
```

In this list, we have the properties of two Linux app services and one Windows app service.

The goal of this recipe is to filter Linux entries and Windows entries from the above list to create corresponding app services.

How to do it...

To filter a list, perform the following steps:

1. In the `main.tf` file, add the following `locals` values:

```
locals {
  linux_web_app   = toset([for each in var.web_apps : each.name if
each.os == "Linux"])
  windows_web_app = toset([for each in var.web_apps : each.name if
each.os == "Windows"])
}
```

2. Then, add the `azurerm_linux_web_app` resource to create Linux app services:

```
resource "azurerm_linux_web_app" "app" {
  for_each = local.linux_web_app

  name                = each.value
  location            = "westeurope"
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id     = azurerm_service_plan.linux-plan-app.id

  site_config {}
}
```

3. Also, add the `azurerm_windows_web_app` resource to create Windows app services:

```
resource "azurerm_windows_web_app" "app" {
  for_each = local.windows_web_app

  name                = each.value
  location            = "westeurope"
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id     = azurerm_service_plan.windows-plan-app.id

  site_config {}
}
```

4. Finally, run the Terraform `init`, `plan`, and `apply` commands to provision these three app services.

How it works...

In *Step 1*, we instantiate two `locals` values; the `linux_web_app` will contain a set of Linux app services from the main list, `web_apps`, and the `windows_web_app` will contain a set of Windows app services from the same list.

The filter operation is in the `locals` value, with the following snippet code:

```
linux_web_app = toset([for each in var.web_apps : each.name if each.
os=="Linux"])
```

The result of `linux_web_app` is a set of objects that is selected by looping over `web_apps` objects, depending on one condition: if the `os` value is Linux.

So the result of this set is `webappdemobook1` and `webapptestbook2`.

We also filter the Windows web app on the `web_apps` list using the following condition: if the os is Windows. The result of this set is `webapptestbook3`.

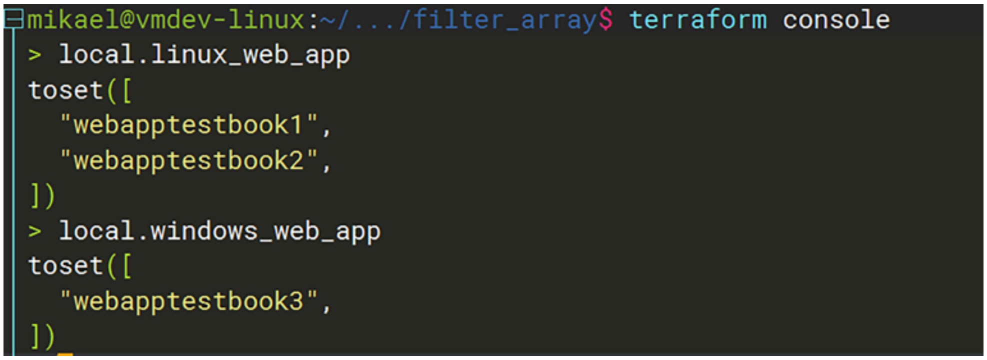
In *Step 2* and *Step 3*, we use these local values, using `for_each` to loop over the sets.

To set the value of the web app name, we use the expression `each.value`, which contains the current element in the set.

There's more...

To preview the result of the local variables `linux_web_app` and `windows_web_app` before applying changes with the `apply` command, we can run the `terraform console` command before `apply`.

The following image shows a preview of the `locals` variables:



```
mikael@vmdev-linux:~/.../filter_array$ terraform console
> local.linux_web_app
toset([
  "webapptestbook1",
  "webapptestbook2",
])
> local.windows_web_app
toset([
  "webapptestbook3",
])_
```

Figure 3.10: Terraform console preview of the locals variables

We can see the content of the `locals` variables `linux_app_app` and `windows_web_app`.

See also

- The documentation of the `toset` function is available here: `https://www.terraform.io/language/functions/toset`.
- We can also sort and group a list, as explained in this documentation: `https://www.terraform.io/language/expressions/for#filtering-elements`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

4

Using Terraform with External Data

When using a Terraform configuration, sometimes we might be required to get information about existing resources or infrastructures that were not provisioned by Terraform. Moreover, we may need to interact with the local system by manipulating local files or even running programs that are installed on the local system.

In this chapter, we will learn how to retrieve data from external systems by using data sources and querying external resources. We will cover the use of Terraform for local operations, such as running a local executable and manipulating local files. Finally, we will learn how to use the Terraform Shell provider to execute shell scripts.

In the past, I have seen some people trying to build their whole infrastructure (entirely managed by Terraform) just using resources and data sources. This is usually not a good practice.

The reason it's not a good practice is that it's comparable to having two different apps that communicate by accessing each others' databases, as opposed to public APIs. We can think of Terraform outputs as explicit public APIs and state as internal state (akin to a database).

There is also a need for at least read permissions to each relevant resource – just to consume its state. Some fields may not ever be available via dedicated data sources, such as database passwords that are only provided upon creation and stored in the state only.

Finally, unlike outputs, data sources are more prone to breaking changes as a result of provider upgrades.

So use data sources, but with caution.

In this chapter, we will cover the following recipes:

- Obtaining external data with data sources
- Querying external data with Terraform
- Manipulating local files with Terraform
- Executing local programs with Terraform
- Executing shell scripts using the Terraform Shell provider

Let's get started!

Technical requirements

For this chapter, you will need to have the Terraform binary installed on your computer and some basic knowledge of scripting with PowerShell or Shell.

The source code for this chapter is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP04`.

Obtaining external data with data sources

When infrastructure is provisioned with Terraform, it is sometimes necessary to retrieve information about existing resources. Indeed, when deploying resources to a certain infrastructure, there is often a need to place them in an existing infrastructure or link them to other resources that have already been provisioned.

In this recipe, we will learn how, in our Terraform configuration, to retrieve information about resources already present in an infrastructure.

While data sources provide easy access to any resources, they should be primarily used for resources managed outside of Terraform. Resources already managed by Terraform (whether by you or someone else) should instead expose data via outputs, which can then be consumed using the built-in `terraform_remote_state` data source (or `tfe_outputs`), as opposed to Azure data sources.

Getting ready

For this recipe, we will use an existing Terraform configuration that provides an Azure App Service instance in the Azure cloud. This source code is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP04/data`.

This code is incomplete because, for this project, we need to store the App Service instance in an existing App Service plan. This Service Plan is the one we will use for the entire App Service instance.

So for this recipe, we take into consideration that the App Service Plan `app-service-plan` is already provisioned on the resource group `rg-service_plan`.

How to do it…

To create an App Service instance on an existing App Service Plan, perform the following steps:

1. In the file that contains our Terraform configuration, add the following `data` block:

   ```
   data "azurerm_app_service_plan" "myplan" {
     name                = "app-service-plan"
     resource_group_name = "rg-service_plan"
   }
   ```

   In the `properties` block, specify the name and the resource group of the Service Plan to be used.

2. Then, complete the existing App Service configuration, as follows:

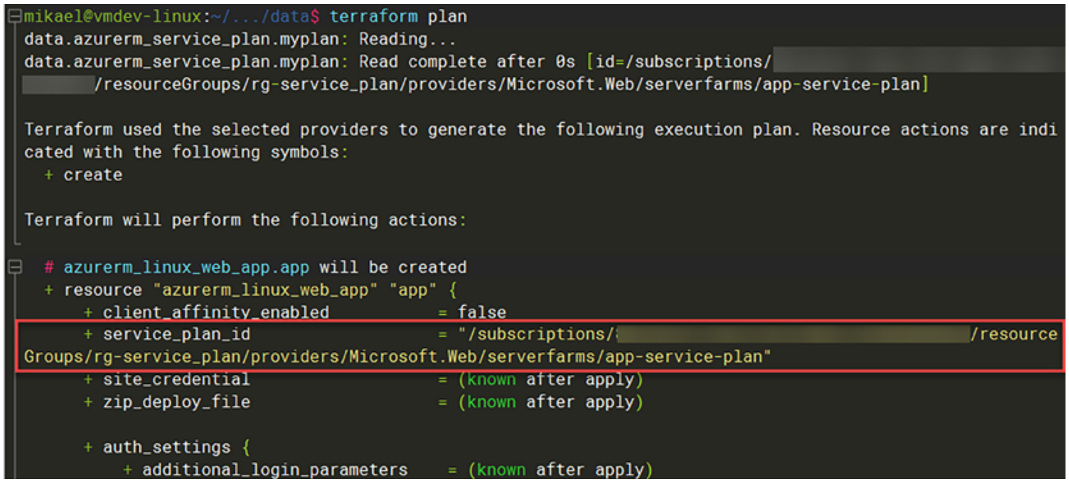
   ```
   resource "azurerm_linux_web_app" "app" {
     name                = "${var.app_name}-${var.environement}"
     location            = azurerm_resource_group.rg-app.location
     resource_group_name = azurerm_resource_group.rg-app.name
     service_plan_id     = data.azurerm_app_service_plan.myplan.id
   }
   ```

How it works…

In *Step 1*, a `data` block is added to query existing resources. In this `data` block, we specify the resource group and the name of the existing Service Plan.

In *Step 2*, we use the ID of the Service Plan that was retrieved by the `data` block we added in *Step 1*.

The result of executing this Terraform configuration can be seen in the following screenshot:



```
mikael@vmdev-linux:~/.../data$ terraform plan
data.azurerm_service_plan.myplan: Reading...
data.azurerm_service_plan.myplan: Read complete after 0s [id=/subscriptions/[redacted]
[redacted]/resourceGroups/rg-service_plan/providers/Microsoft.Web/serverfarms/app-service-plan]

Terraform used the selected providers to generate the following execution plan. Resource actions are indi
cated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_linux_web_app.app will be created
  + resource "azurerm_linux_web_app" "app" {
      + client_affinity_enabled           = false
      + service_plan_id                   = "/subscriptions/[redacted]/resource
Groups/rg-service_plan/providers/Microsoft.Web/serverfarms/app-service-plan"
      + site_credential                   = (known after apply)
      + zip_deploy_file                   = (known after apply)

      + auth_settings {
          + additional_login_parameters    = (known after apply)
```

Figure 4.1: Terraform data source information

As we can see, we have the ID of the Service Plan that was retrieved by the `data` block.

`terraform plan` will indeed retrieve data sources and show most of the values, but one other (perhaps more reliable) way is to use `terraform apply` and `terraform show` afterward.

The main difference is that `apply` will save the data to the state file and `show` will display the data from the state file on demand Users typically use show instead of plan to inspect the state file. [‘show’ and ‘plan’ would be styled as code].

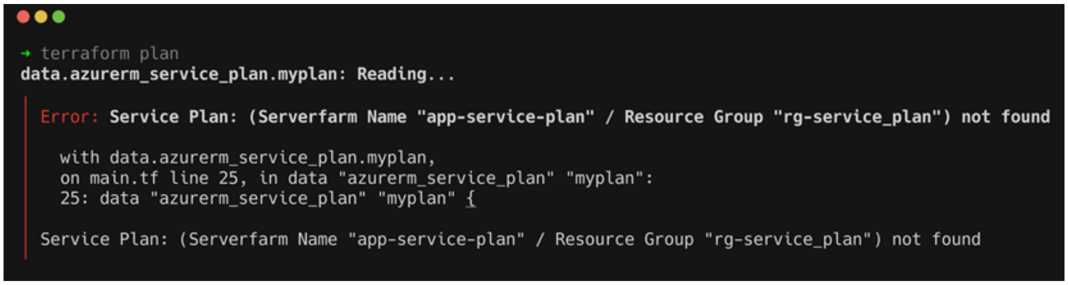
There’s more...

What’s interesting about the use of `data` blocks is that when executing the `terraform destroy` command on our Terraform configuration, Terraform does not perform a destroy action on the resource called by the `data` block.

Moreover, the use of `data` blocks is to be preferred to the use of IDs written in clear text in the code, which can change, because the `data` block recovers the information dynamically.

Finally, the `data` block is also called when executing the `terraform plan` command, so your external resource must be present before you execute the `terraform plan` and `terraform apply` commands.

If this external resource is not already present, we get the following error in the `terraform plan` command:



```
→ terraform plan
data.azurerm_service_plan.myplan: Reading...

│ Error: Service Plan: (Serverfarm Name "app-service-plan" / Resource Group "rg-service_plan") not found
│
│   with data.azurerm_service_plan.myplan,
│   on main.tf line 25, in data "azurerm_service_plan" "myplan":
│   25: data "azurerm_service_plan" "myplan" {
│
│ Service Plan: (Serverfarm Name "app-service-plan" / Resource Group "rg-service_plan") not found
```

Figure 4.2: Terraform data source error

You need to know which providers to use in your Terraform configuration since not all providers implement `data` blocks.

We will learn how to get resource information from other Terraform state files in the recipe *Using external resources from other Terraform state files* in *Chapter 5*, *Managing Terraform State*.

See also

For more information about `data` blocks, see the following documentation: `https://www.terraform.io/docs/configuration/data-sources.html`.

Querying external data with Terraform

In the previous recipe, we learned that it is possible to use the `data` block to retrieve external data. However, there are scenarios where the `data` block does not exist in the provider, such as when we need to talk with an external API or use a local tool and process its output.

To meet this need, the `external` resource in Terraform that allows you to call an external program and retrieve its output data so that it can be used in the Terraform configuration.

Use of the `external` provider imposes prerequisites that may not be obvious (for example, in this case, we expect a particular version of PowerShell) or may be difficult to communicate other than through README files or documentation. Also, Terraform is generally designed to work the same cross-platform (operating system/architecture), but this essentially restricts the configuration to platforms that can (and do) run PowerShell. These requirements apply to both CI and local environments.

In this recipe, we will learn how to call an external program and retrieve its output so that we can reuse it.

Getting ready

For this recipe, we will use an existing Terraform configuration that allows us to provision a resource group in Azure.

To perform this recipe, we need to install PowerShell Core. The installation documentation detailing how to do this is available here:

Here, we want a Azure Resource Group to be in a different Azure region (location), depending on the environment (development or production).

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP04/external`.

How to do it...

Perform the following steps:

1. In the directory that contains our `main.tf` file, create a PowerShell `GetLocation.ps1` script that contains the following content:

```
# Read the JSON payload from stdin
$jsonpayload = [Console]::In.ReadLine()

# Convert JSON to a string
$json = ConvertFrom-Json $jsonpayload
$environment = $json.environment

if($environment -eq "Production"){
 $location="westeurope"
}else{
 $location="westus"
}

# Write output to stdout
Write-Output "{ ""location"" : ""$location""}"
```

2. In the `main.tf` file, add the `external` block, as follows:

   ```
   data "external" "getlocation" {
     program = ["pwsh", "./GetLocation.ps1"]
     query = {
         environment = "${var.environment_name}"
     }
   }
   ```

3. Then, modify the code of the resource group to make its location more dynamic, as follows:

   ```
   resource "azurerm_resource_group" "rg" {
     name = "RG-${local.resource_name}"
     location = data.external.getlocation.result.location
   }
   ```

4. Finally, we add an `output` value that has the following configuration:

   ```
   output "locationname" {
     value = data.external.getlocation.result.location
   }
   ```

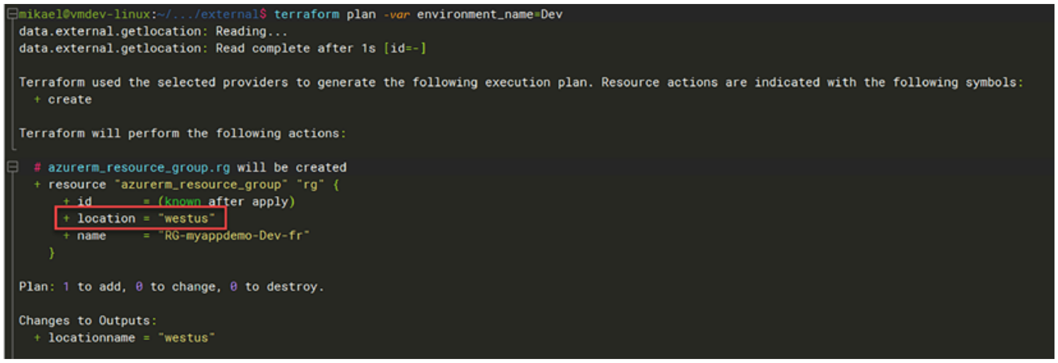
How it works...

In *Step 1*, we wrote the PowerShell `GetLocation.ps1` script, which will be called by Terraform locally. This script takes in `environment` as an input parameter in JSON format. Then, this PowerShell script makes a condition on this input environment and returns the right Azure region as output so that we can use it in our Terraform configuration.

We used the Terraform `external` resource, which calls this PowerShell script and provides it with the contents of the `environment_name` variable as a parameter.

Then we add the return value of this `data` block with an `external` data source in the `location` property of the resource group.

The following screenshot shows the output of executing `terraform plan` with the `environment_name` variable, which is set to `Dev`:

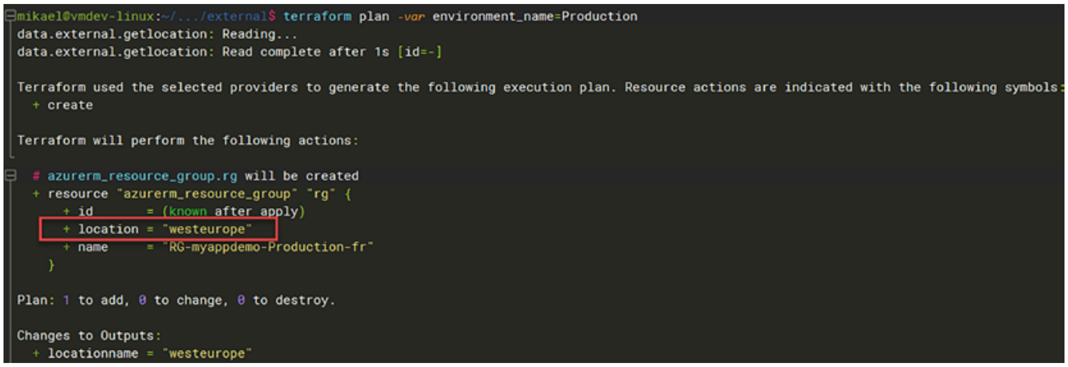


```
mikael@vmdev-linux:~/.../external$ terraform plan -var environment_name=Dev
data.external.getlocation: Reading...
data.external.getlocation: Read complete after 1s [id=-]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id       = (known after apply)
      + location = "westus"
      + name     = "RG-myappdemo-Dev-fr"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + locationname = "westus"
```

Figure 4.3: Terraform external data with the Dev variable

As you can see, the regional location of the resource group is `westus`.

The following screenshot shows the output executing `terraform plan` with the `environment_name` variable, which is set to `Production`:



```
mikael@vmdev-linux:~/.../external$ terraform plan -var environment_name=Production
data.external.getlocation: Reading...
data.external.getlocation: Read complete after 1s [id=-]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "RG-myappdemo-Production-fr"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + locationname = "westeurope"
```

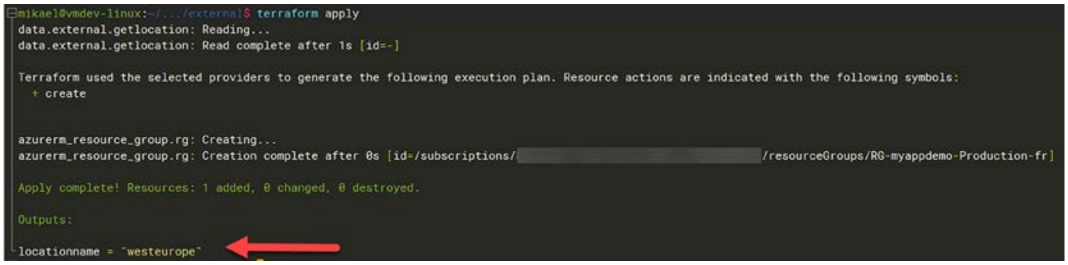
Figure 4.4: Terraform external data with the Production variable

As you can see, the location of the resource group is `westeurope`.

In *Chapter 2*, *Writing Terraform Configurations*, in the recipe *Manipulating variables*, we used the `-var` option of the `terraform plan` command, which allowed us to assign a value to a variable upon executing the command.

Finally, in *Step 3*, we add a Terraform `output` that exposes this value. This can be displayed upon executing Terraform. This can also be exploited at other places in the Terraform configuration.

The following screenshot shows the output after running the `terraform apply` command:



```
mikael@vmdev-linux:~/.../external$ terraform apply
 data.external.getlocation: Reading...
 data.external.getlocation: Read complete after 1s [id=-]

 Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
   + create


 azurerm_resource_group.rg: Creating...
 azurerm_resource_group.rg: Creation complete after 0s [id=/subscriptions/[redacted]/resourceGroups/RG-myappdemo-Production-fr]

 Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

 Outputs:

 locationname = "westeurope"
```

Figure 4.5: Terraform output of external data

As we can see, the `terraform output` command displays the right `locationname` value.

There's more...

In this recipe, we used a PowerShell script, but this script also works with all the other scripting languages and tools that are installed on your local machine.

The `external` resource contains specifics about the protocol, the format of the parameters, and its output. I advise that you read its documentation to learn more: `https://registry.terraform.io/providers/hashicorp/external/latest/docs/data-sources/external`.

We will learn another solution to running local scripts using the Terraform Shell provider, in the *Executing shell scripts using the Terraform Shell provider* recipe of this chapter.

Additionally, we can also use the `TerraCurl` Terraform provider to call any REST API, to post/delete data, and also to get data and use this data in Terraform output. For more information read the article here: `https://www.hashicorp.com/blog/writing-terraform-for-unsupported-resources`.

See also

The following are a couple of articles with examples of how to use the `external` Terraform resource:

- `https://dzone.com/articles/lets-play-with-terraform-external-provider`

Manipulating local files with Terraform

Terraform is very popular due to its infrastructure as code functionality for cloud providers. But it also has many providers that allow us to manipulate the local system.

In the *Querying external data with Terraform* recipe of this chapter, we discussed local script executions that are performed by Terraform to get data for external data sources.

In this recipe, we will study another type of local operation that involves creating and archiving local files with Terraform.

Getting ready

For this recipe, we don't need any prerequisites or base code – we will write the configuration from scratch.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP04/files` and `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP04/archive`.

How to do it…

Perform the following steps:

1. In a new folder called `files`, create a `main.tf` file. Write the following code inside it:

```
resource "local_file" "myfile" {
  content = "This is my text"
  filename = "../mytextfile.txt"
}
```

2. In a command-line terminal, navigate to the `files` directory and execute Terraform's workflow commands, which are as follows:

```
terraform init
terraform plan
terraform apply
```

3. In a new `archive` folder, create a `main.tf` file and write the following Terraform configuration inside it:

```
data "archive_file" "backup" {
  type = "zip"
  source_file = "../mytextfile.txt"
  output_path = "${path.module}/archives/backup.zip"
}
```

4. Then, using the command-line terminal, navigate to the `archive` directory and execute the following Terraform commands:

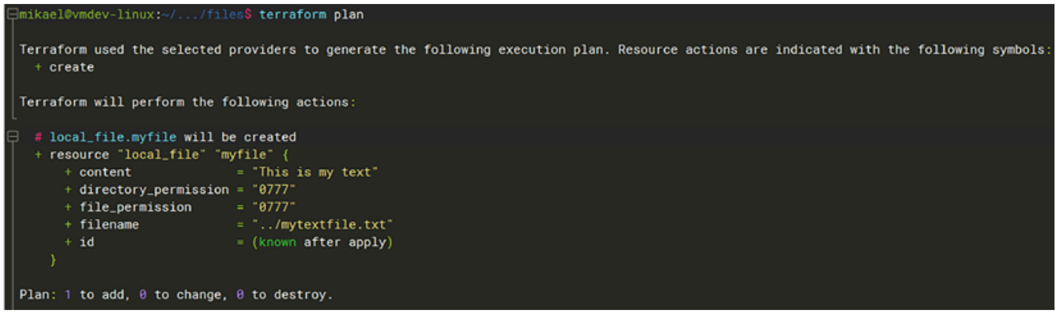
```
terraform init
terraform plan
```

How it works...

In *Step 1*, we wrote part of a Terraform configuration that uses the `local` provider and the `local_file` resource. This resource creates a file called `mytextfile.txt` and adds `This is my text` to it.

Then, in *Step 2*, we executed Terraform on this code. By doing this, we created the `mytextfile.txt` file on our local disk.

The result of executing the `terraform plan` command on this code can be seen in the following screenshot:



```
mikael@vmdev-linux:~/.../files$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # local_file.myfile will be created
  + resource "local_file" "myfile" {
      + content              = "This is my text"
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "../mytextfile.txt"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Figure 4.6: Terraform local file provider

After we executed `terraform apply`, the `mytextfile.txt` file became available on our local filesystem.

In the second part of this recipe, in *Step 3*, we wrote a part of Terraform configuration that uses the `archive` provider and the `archive_file` resource to create a ZIP file that contains the file we created in *Steps 1* and *2*.

After we executed `terraform apply`, the ZIP archive `backup.zip` file became available on our local filesystem, in the `archives` folder.

There's more...

As we can see, the `archive_file` resource we used in the second part of this recipe is of the `data` block type (which we learned about in the *Obtaining external data with data sources* recipe of this chapter) and is therefore based on an element that already existed before we executed the `terraform plan` command.

In our case, the file to be included in the archive must already be present on the local disk.

See also

- Documentation on the `local_file` resource is available at `https://registry.terraform.io/providers/hashicorp/local/latest/docs/resources/file`.
- Documentation on the `archive_file` resource is available at `https://registry.terraform.io/providers/hashicorp/archive/latest/docs/data-sources/file`.

Executing local programs with Terraform

As we saw in the previous recipe regarding file manipulation, apart from infrastructure provisioning, Terraform also allows you to run programs or scripts that are located on the local workstation where Terraform has been installed.

In this recipe, we will learn how to execute a local program inside a Terraform configuration.

Getting ready

For this recipe, we will complete the Terraform configuration that we used in the previous recipe to write a file on the local machine. Our goal will be to execute a PowerShell command with Terraform that will read and display the contents of the file that we have written using Terraform.

> The technical requirement for this recipe is that we will run this Terraform script on a Windows operating system. You can of course adapt it to run programs on Linux or Mac operating systems or use the Terraform Shell provider, which we will learn about in the next recipe.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP04/files_local_exec`.

How to do it…

Perform the following steps:

1. In the `main.tf` file, which is in the `files` directory of the source code from the previous recipe, complete the Terraform configuration with the following code:

```
resource "null_resource" "readcontentfile" {
  provisioner "local-exec" {
    command = "Get-Content -Path ../mytextfile.txt"
    interpreter = ["pwsh", "-Command"]
  }
}
```

2. Then, in a command-line terminal, execute the Terraform workflow commands, as follows:

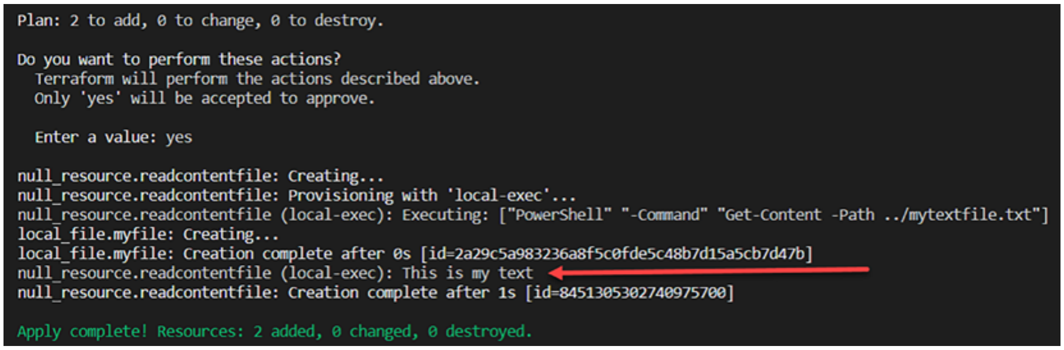
```
terraform init
terraform plan
terraform apply
```

How it works...

In this recipe, we used `null_resource`, which is a `null` provider resource. This resource doesn't allow us to create resources, but rather to run programs locally.

In this resource, we have the `provisioner` block, which is of the `local-exec` type, which operates on our local machine. Then, in this block, we indicate the command to execute, which is the `Get-Content` command of PowerShell. With this, we are telling Terraform to use the PowerShell interpreter to execute this command.

When executing the respective Terraform commands, we get the following result:



```
Plan: 2 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

null_resource.readcontentfile: Creating...
null_resource.readcontentfile: Provisioning with 'local-exec'...
null_resource.readcontentfile (local-exec): Executing: ["PowerShell" "-Command" "Get-Content -Path ../mytextfile.txt"]
local_file.myfile: Creating...
local_file.myfile: Creation complete after 0s [id=2a29c5a983236a8f5c0fde5c48b7d15a5cb7d47b]
null_resource.readcontentfile (local-exec): This is my text
null_resource.readcontentfile: Creation complete after 1s [id=8451305302740975700]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

Figure 4.7: Terraform local program executed

As you can see, the text `This is my text`, which we had written in the file (in the `local_file` resource), is displayed in the Terraform runtime output.

There's more...

In this recipe, we looked at a simple `local-exec` command being executed with Terraform. It is also possible to execute several commands that are stored in a script file (Bash, PowerShell, and so on) with a sample Terraform configuration (which works on Windows and can be adapted for Linux), as shown here:

```
resource "null_resource" "readcontentfile" {
  provisioner "local-exec" {
    command = "myscript.ps1"
    interpreter = ["pwsh", "-Command"]
  }
}
```

The `local-exec` provisioner sets expectations on the local system, which may not be obvious. This is usually otherwise mitigated by cross-platform builds from providers and Terraform itself, where the implementation should generally work the same on any supported platform (macOS/Linux/Windows).

In addition, it is important to know that the `local-exec` provisioner, once executed, ensures that the Terraform state file cannot be executed a second time by the `terraform apply` command.

To be able to execute the `local-exec` command based on a trigger element, such as a resource that has been modified, it is necessary to add a `trigger` object inside `null_resource` that will act as the trigger element of the `local-exec` resource.

The following example code uses a trigger, based on `timestamp`, to execute the `local-exec` code at each execution step of Terraform:

```
resource "null_resource" "readcontentfile" {
  triggers = {
   trigger = timestamp()
  }
  provisioner "local-exec" {
   command = "Get-Content -Path ../mytextfile.txt"
   interpreter = ["PowerShell", "-Command"]
  }
}
```

In this example, the trigger is a timestamp that will have a different value each time Terraform is run.

We will look at another concrete use case of `local-exec` in the *Executing Azure CLI commands in Terraform* recipe in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*.

See also

The `local-exec` provisioner documentation is available at `https://www.terraform.io/docs/provisioners/local-exec.html`.

Executing shell scripts using the Terraform Shell provider

In the previous recipe, we learned that it is possible to execute local programs or commands with Terraform by using the `null_resource` resource.

There is another solution to execute shell scripts to manage external resources by using local commands or an API call.

The solution that we will discuss in this recipe is the use of the Terraform Shell provider in a simple scenario.

Getting ready

To complete this recipe, you'll need only to have some knowledge about Shell scripting.

This recipe can be applied natively on Linux and macOS operating systems. To perform this recipe on Windows, you can use WSL. The documentation for WSL is available here:

In this recipe, we will mock an API call that manages a book store. We will only use the JSON content created locally in the `book.json` file.

The complete source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP04/shell`.

How to do it…

To use the Terraform Shell provider, we perform the following steps:

1. In the new `main.tf` file, write the following content:

```
terraform {
  required_providers {
    shell = {
      source = "scottwinkler/shell"
```

```
        version = "1.7.10"
      }
    }
  }

  provider "shell" {}
```

2. Then, we use the `shell_script` resource with the following content:

```
resource "shell_script" "sh" {
    lifecycle_commands {
        create = file("${path.module}/scripts/create.sh")
        read   = file("${path.module}/scripts/read.sh")
        delete = file("${path.module}/scripts/delete.sh")
    }

    interpreter = ["/bin/bash", "-c"]
    working_directory = path.module
    triggers = {
        timestamp = timestamp()
    }
}

output "id" {
    value = shell_script.sh.output["id"]
}
```

3. In the new subfolder, `scripts`, create the script `create.sh` with the content:

```
/bin/cat <<END >book.json
  {"id": "1", "title": "Terraform Cookbook", "Author": "MK", "tags":
"terraform-Azure"}
END
cat book.json
```

   The complete source code of this script is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP04/shell/scripts/create.sh`.

4. Write the `read.sh` file with the following content:

```
cat book.json
```

The complete source code of this script is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP04/shell/scripts/read.sh`.

5. Write `delete.sh` with the following content:

```
rm -rf book.json
```

The complete source code of this script is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP04/shell/scripts/delete.sh`.

6. Finally, run the Terraform workflow with the `init`, `plan`, and `apply` commands.

How it works...

In *Step 1*, we instantiate the Shell provider by declaring the `source`, the `version`, and the provider configuration, which is empty for our sample.

In *Step 2*, we use the Shell provider with the following configuration:

- The `shell_script` resource, which calls three scripts: one for the `create` operation that is called during the `apply` Terraform command, one for the `read` operation that is called in the `plan` Terraform command, and one for the `delete` operation that is called during the `destroy` Terraform command.
- We also specify the interpreter (which is `bash`) and the trigger to run this resource at each use of `terraform apply` (exactly like the `null_resource` resource).
- The output that returns the JSON `Id` property of the created book.

The documentation of the `shell_script` resource is available here: `https://registry.terraform.io/providers/scottwinkler/shell/latest/docs/resources/shell_script_resource`.

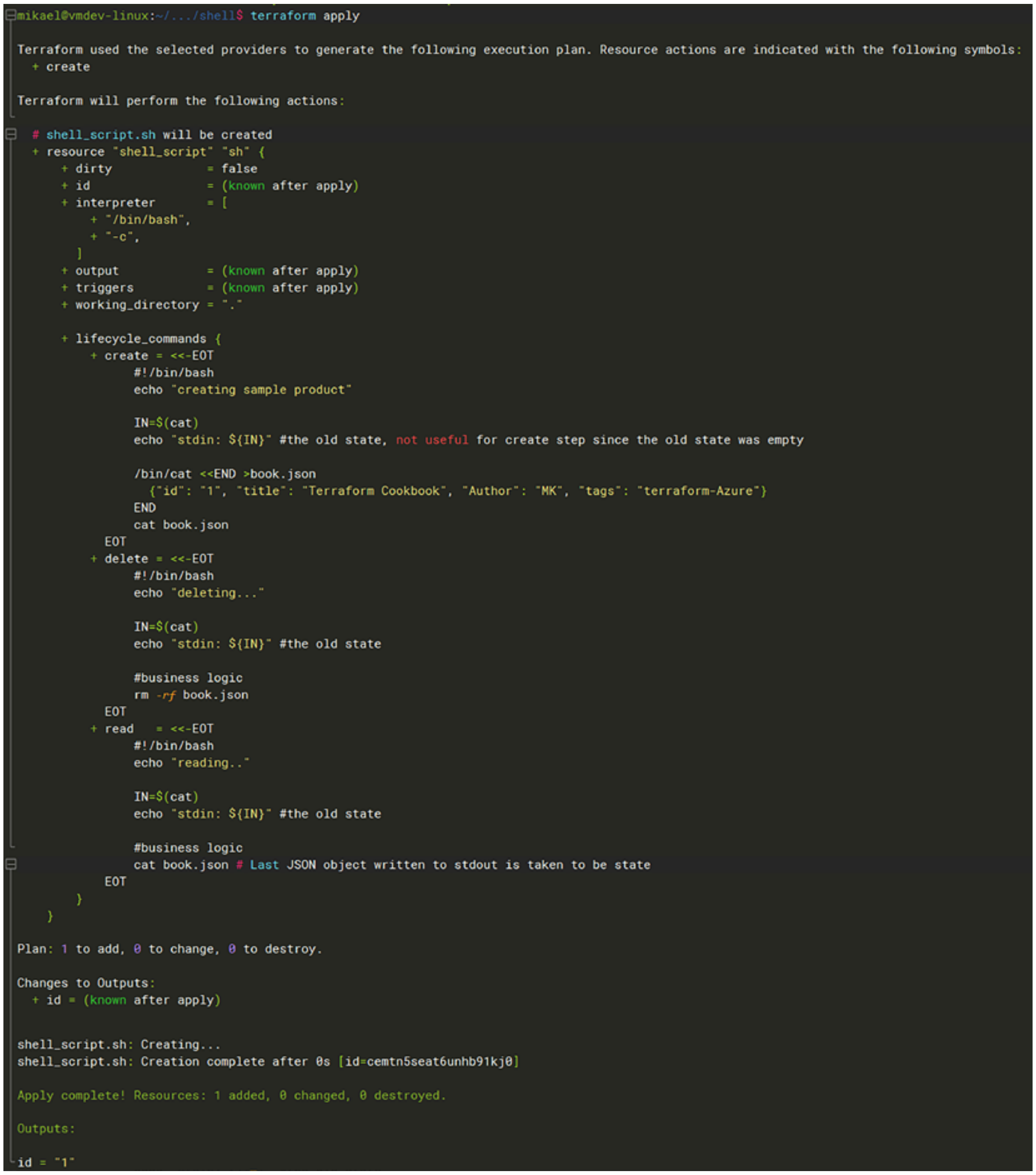
Then, in *Step 3*, we write the shell script `create.sh`, which creates the `book.json` file, which contains the JSON content of the created book (for the mock); this script will be executed by the `apply` command of Terraform.

In *Step 4*, we write the `read.sh` script, which reads the `book.json` file created on the `plan` command.

In *Step 5*, we write the `delete.sh` script, which deletes the `book.json` file; this script will be executed by the `destroy` command.

Finally, in *Step 6*, we run the Terraform workflow commands with `init`, `plan`, and `apply`.

The following image shows the `terraform apply` command:



```
mikael@vmdev-linux:~/.../shell$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # shell_script.sh will be created
  + resource "shell_script" "sh" {
      + dirty             = false
      + id                = (known after apply)
      + interpreter       = [
          + "/bin/bash",
          + "-c",
        ]
      + output            = (known after apply)
      + triggers          = (known after apply)
      + working_directory = "."

      + lifecycle_commands {
          + create = <<-EOT
                #!/bin/bash
                echo "creating sample product"

                IN=$(cat)
                echo "stdin: ${IN}" #the old state, not useful for create step since the old state was empty

                /bin/cat <<END >book.json
                  {"id": "1", "title": "Terraform Cookbook", "Author": "MK", "tags": "terraform-Azure"}
                END
                cat book.json
            EOT
          + delete = <<-EOT
                #!/bin/bash
                echo "deleting..."

                IN=$(cat)
                echo "stdin: ${IN}" #the old state

                #business logic
                rm -rf book.json
            EOT
          + read   = <<-EOT
                #!/bin/bash
                echo "reading.."

                IN=$(cat)
                echo "stdin: ${IN}" #the old state

                #business logic
                cat book.json # Last JSON object written to stdout is taken to be state
            EOT
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + id = (known after apply)


shell_script.sh: Creating...
shell_script.sh: Creation complete after 0s [id=cemtn5seat6unhb91kj0]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

id = "1"
```

Figure 4.8: Terraform Shell provider execution

We can see the Terraform output and the newly created file, `book.json`, at the root of the Terraform configuration directory.

There's more...

In this recipe, we used the Shell provider with the `create`, `read`, and `delete` operations. We can also add the `update` operation, which will be applied by the `apply` command if the content of the JSON file is changed.

The Terraform Shell provider also contains the data source `shell_script`, which only does the `read` operation. For more information, read the documentation here: `https://registry.terraform.io/providers/scottwinkler/shell/latest/docs/data-sources/shell_script`.

The `keepers = timestamp()` line in the configuration shouldn't be necessary. There may be scenarios where it's relevant, but in general, if the "read" script always provides the data, which reflects the reality, then this will be sufficient to detect a change and generate a *diff* to show us these changes. Bypassing the diff detection and planning part does somewhat call into question whether the problem is best solved in Terraform, since the planning part is one of its key features. At this point, Terraform arguably just becomes just a program that runs another program. However, we have chosen to keep all the infrastructure pieces together in one configuration so you can see all the parts and how they fit together. It is your call as to whether you wish to modify the code or keep it all together in one configuration.

Additionally, any and all Bash/shell scripts can generally be made more flexible with a change to the top shebang notation, which makes it more likely to work on more Linux distributions. If this is something you are curious about testing yourself, see `https://stackoverflow.com/questions/16365130/what-is-the-difference-between-usr-bin-env-bash-and-usr-bin-bash`.

See also

- The documentation of the Terraform Shell provider is available here: `https://registry.terraform.io/providers/scottwinkler/shell/latest/docs`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

5

Managing Terraform State

One of the most important artifacts of Terraform is the Terraform state file. Terraform state is the file that contains all the configuration that has been applied during the Terraform workflow.

> Since the Terraform State file is not necessarily a physical file but can also be in blob object format, for example, throughout this chapter, we will use the term Terraform State to refer to the Terraform State file.
>
> For more information about Terraform State file, read the documentation overview here: `https://developer.hashicorp.com/terraform/language/state`.

In most situations, we don't need to interact with it directly. However, you may need to manipulate it either to view its information, delete resources, import resources, or even move resources. Knowing that this file is a JSON file, we can be tempted to manipulate it manually. This is not a good practice, as it may cause errors that may make it impossible to run Terraform afterward. Therefore, it is recommended to use the tools provided by HashiCorp, such as the Terraform CLI or even the specific Terraform configuration block.

In this chapter, we will learn how to manipulate Terraform State safely using the Terraform CLI to list resources, delete resources, synchronize resources, import existing resources, and move resources. We will also learn how to use external resources from other state files and refactor the Terraform configuration using the Terraform `moved` block.

We will cover the following recipes in this chapter:

- Using the local Terraform State

- Managing resources in Terraform state
- Synchronizing Terraform state
- Importing existing resources
- Using external resources from other Terraform state files
- Refactoring resources in configuration

Using the local Terraform state

In the default behavior of Terraform, if no backend configuration is provided, Terraform state is stored locally and is named `terraform.tfstate`.

This file is stored in the same folder that contains the Terraform configuration. (This folder is also called the root module configuration.)

This is an approach to managing local state when starting with Terraform or for a proof-of-concept project.

In this recipe, we will learn how to configure the local Terraform state.

Getting ready

This recipe doesn't have any prerequisites; we will just generate a password with Terraform using the following configuration:

```
resource "random_password" "password" {
  length           = 16
  special          = true
  override_special = "_%@"
}
```

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/localstate`.

How to do it…

To configure the local Terraform state, in the `main.tf` file, write the following Terraform configuration:

```
terraform {
  backend "local" {
    path = "../../demo.tfstate"
  }
}
```

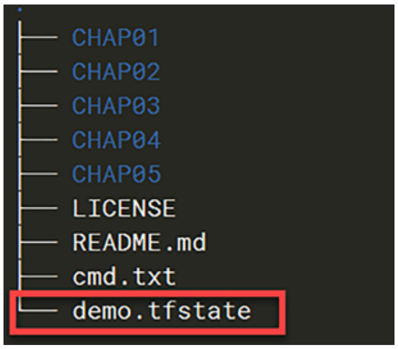
The complete source configuration is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP05/localstate/main.tf`.

Then, we run the Terraform workflow with `init`, `plan`, and `apply` commands.

How it works…

In the `main.tf` file, we write the configuration of the `backend` block, which is a `local` file, and in the `path` property, we define the relative path of the state file with `../../demo.tfstate`.

Then, we run the `init`, `plan`, and `apply` commands, and we can see the `demo.tfstate` file in the specified folder (`path = "../../demo.tfstate"`):



```
.
├── CHAP01
├── CHAP02
├── CHAP03
├── CHAP04
├── CHAP05
├── LICENSE
├── README.md
├── cmd.txt
└── demo.tfstate
```

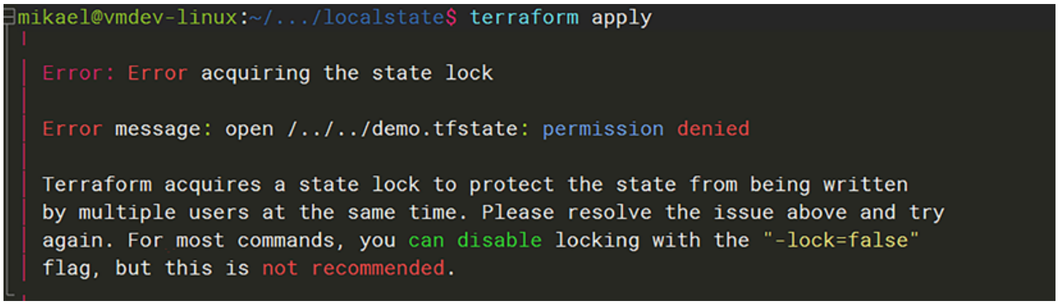
Figure 5.1: Local Terraform state

We can see that the `demo.tfstate` file is created at the specified path (`../../`).

There's more…

About the local Terraform Path; in the `backend` configuration, we can't configure the absolute path of the state file, only the relative path. Additionally, the Terraform CLI must have the necessary permissions to create and write files in the target folder.

If Terraform doesn't have permission, we will get this error when running `apply`:



```
mikael@vmdev-linux:~/.../localstate$ terraform apply

Error: Error acquiring the state lock

Error message: open /../../demo.tfstate: permission denied

Terraform acquires a state lock to protect the state from being written
by multiple users at the same time. Please resolve the issue above and try
again. For most commands, you can disable locking with the "-lock=false"
flag, but this is not recommended.
```

Figure 5.2: Permission denied on local Terraform state

To resolve this error, check and provide permission in the target folder.

If we want to use this local state file as a remote data state, we can write the following configuration:

```
data "terraform_remote_state" "test" {
  backend = "local"

  config = {
    path = "${path.module}/../../demo.tfstate"
  }
}
```

We will learn more details about this in the *Using external resources from other Terraform state files* recipe in this chapter.

The problems with storing the state file alongside the folder that contains the Terraform configuration are as follows:

Terraform state can contain sensitive information.

While Terraform state contains the configuration (which itself is human-readable and versioned), it also contains other metadata sourced from remote APIs and essentially acts as an internal database for Terraform. Just like storing binaries in VCS is not a common practice because comparing changes is meaningless in most cases, storing state in VCS is equally discouraged for similar reasons. It can't be accessed by other Terraform configurations, which must access our state remotely.

Finally, to resolve the issues above, one of the best practices with Terraform is to use a remote backend and not to use the `local` backend because it isn't secure. We will see an example of using a remote backend in Azure in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*. So, before using the local state file, analyze your infrastructure system and use the remote backend state.

See also

The documentation of the local backend is available here: `https://developer.hashicorp.com/terraform/language/settings/backends/local`.

Managing resources in Terraform state

One important practice to take into consideration when we use Terraform is to never edit the Terraform state file manually. Even if it is a JSON file, any manual modification incurs a risk of corrupting it and thus making it unavailable for resource deployment.

In addition, companies should make certain that the users who apply the Terraform configurations have permission to access the backend files that contain state files.

In response to the necessity of certain scenarios to display or update Terraform state, Terraform has a set of commands that allow you to manage state safely.

In this recipe, we will learn how to manage Terraform state by detailing some of the Terraform commands dedicated to Terraform state.

Let's get started!

Getting ready

To complete this recipe, we will suppose that we have already provisioned resources in Azure with Terraform using the following configuration:

```
resource "azurerm_resource_group" "rg-app" {
  name     = "${var.resource_group_name}-${var.environment}"
  location = var.location
}

resource "azurerm_service_plan" "plan-app" {
  name                = "${var.service_plan_name}-${var.environment}"
  location            = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  os_type   = "Windows"
  sku_name = "S1"

}

resource "azurerm_windows_web_app" "app" {
  name                = "${var.app_name}-${var.environment}"
  location            = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id     = azurerm_service_plan.plan-app.id
  site_config {}
}
```

In this recipe, we will perform four operations on Terraform state:

- Display the content of Terraform state
- List Terraform resource names within Terraform state
- Show detailed resource properties in Terraform state
- Delete resources in state

For each of these operations, we will see the necessary Terraform command and its output.

The complete source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/managestate`.

We will use Azure as the Terraform provider, but all commands can be applied to all other providers; these commands aren't specific to one provider.

How to do it...

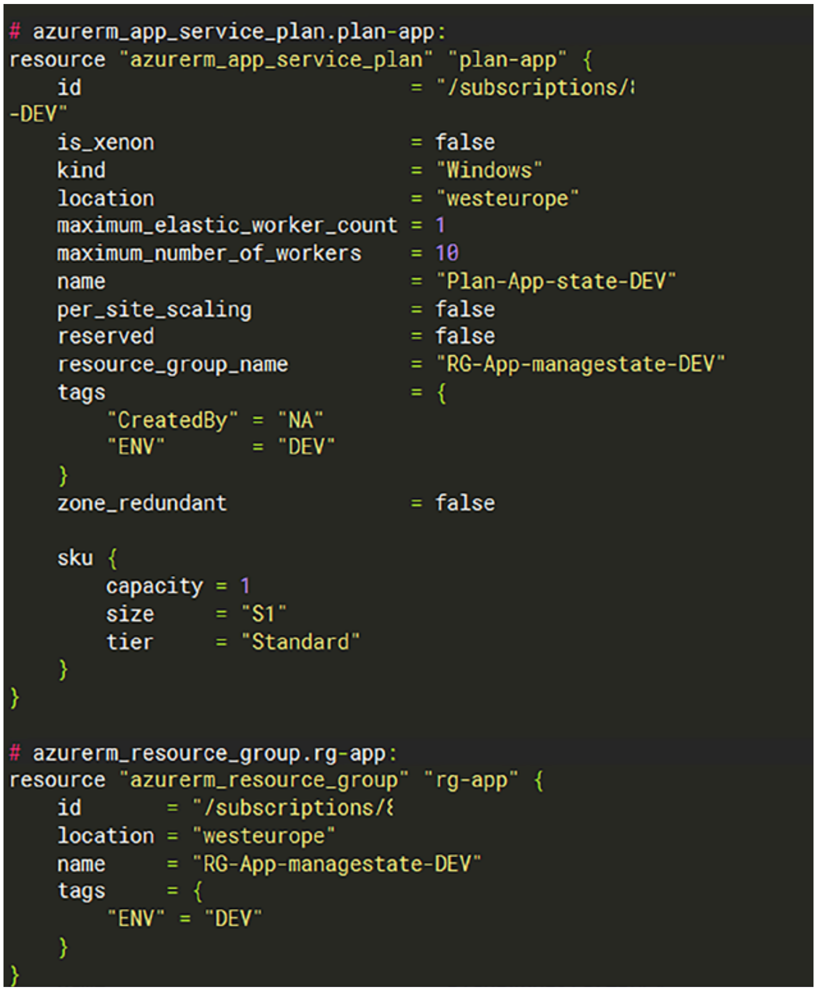
Before running the following commands, we first run the `terraform init`, `plan`, and `apply` commands to create the resources and make sure Terraform is up to date in terms of the configuration.

Displaying the content of state

To display the content of state, run the command:

```
terraform show
```

The following image shows an extract of the output of this command:



```
# azurerm_app_service_plan.plan-app:
resource "azurerm_app_service_plan" "plan-app" {
    id                           = "/subscriptions/
-DEV"
    is_xenon                     = false
    kind                         = "Windows"
    location                     = "westeurope"
    maximum_elastic_worker_count = 1
    maximum_number_of_workers    = 10
    name                         = "Plan-App-state-DEV"
    per_site_scaling             = false
    reserved                     = false
    resource_group_name          = "RG-App-managestate-DEV"
    tags                         = {
        "CreatedBy" = "NA"
        "ENV"       = "DEV"
    }
    zone_redundant               = false

    sku {
        capacity = 1
        size     = "S1"
        tier     = "Standard"
    }
}

# azurerm_resource_group.rg-app:
resource "azurerm_resource_group" "rg-app" {
    id       = "/subscriptions/
    location = "westeurope"
    name     = "RG-App-managestate-DEV"
    tags     = {
        "ENV" = "DEV"
    }
}
```

Figure 5.3: Output sample of the terraform state show command

The output of the `show` command displays the content of state in human-friendly mode with all the details of each resource.

We can also display this output in JSON format with the command `terraform show -json`, and manipulate the JSON to filter or get information by using the `Jq` tool (see the documentation here: `https://stedolan.github.io/jq/manual/`).

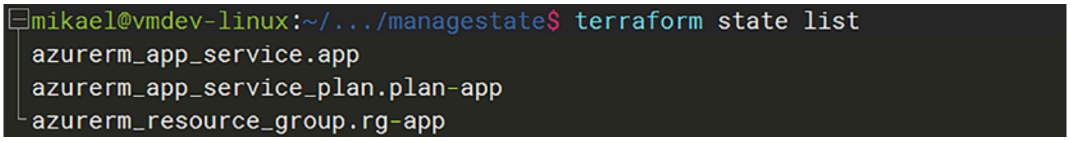
For more information about the `show` command, read the documentation here: `https://developer.hashicorp.com/terraform/cli/commands/show`.

Listing Terraform resource names within state

To list the names of the Terraform resources stored in state, use the following command:

```
terraform state list
```

The following image shows an extract of the output of the command `terraform state list`:



```
mikael@vmdev-linux:~/.../managestate$ terraform state list
azurerm_app_service.app
azurerm_app_service_plan.plan-app
azurerm_resource_group.rg-app
```

Figure 5.4: Sample output of the terraform state list command

We see the list of names of Terraform resources provisioned and written in state.

We can also filter resources by running the command `terraform state list <name>`, for example, to get all resources from a specified array or from a specified module.

In a Shell script, we can also filter the returned list by using the `grep` command, for example:

```
terraform state list | grep '^azurerm_resource_group'
```

The execution of this command returns only the list of resources of the `azurerm_resource_group` type.

For more information about the `state list` command, read the documentation here: `https://developer.hashicorp.com/terraform/cli/commands/state/list`.

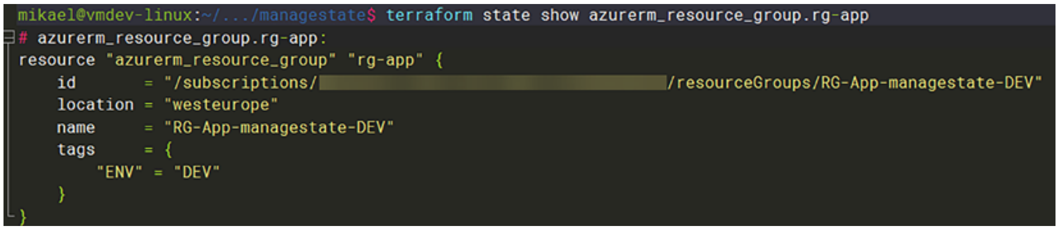
Showing detailed resource properties in state

After displaying the list of resource names, we can display all details about one resource stored in state.

To see these details, run the command `terraform state show <resource name>`, for example, we run the command:

```
terraform state show azurerm_resource_group.rg-app
```

The following image shows the output of this command:



```
mikael@vmdev-linux:~/.../managestate$ terraform state show azurerm_resource_group.rg-app
# azurerm_resource_group.rg-app:
resource "azurerm_resource_group" "rg-app" {
    id       = "/subscriptions/[redacted]/resourceGroups/RG-App-managestate-DEV"
    location = "westeurope"
    name     = "RG-App-managestate-DEV"
    tags     = {
        "ENV" = "DEV"
    }
}
```

Figure 5.5: Sample output of the terraform state show command

We can see the properties of the resource `azurerm_resource_group.rg-app` stored in state in the output above.

For more information about the `state show` command, read the documentation here: `https://developer.hashicorp.com/terraform/cli/commands/state/show`.

Deleting resources from state

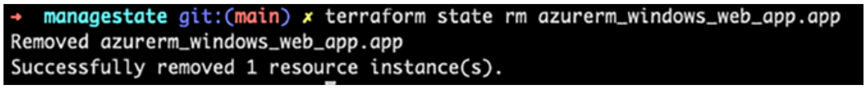
In some cases, we need to remove a resource from state.

> Please note, that this – unlike the `terraform destroy` command – will not remove the underlying resource in Azure; the Azure App Service remains intact. It can be thought of more as "disowning" the management of that resource by Terraform. With that in mind, don't forget to clean up the resource via the Azure portal after removing it from state.

To perform this deletion, run the command `terraform state rm <resource name>`, for example, we run the command:

```
terraform state rm azurerm_windows_web_app.app
```

The following image shows the output of this command:



```
→ managestate git:(main) ✗ terraform state rm azurerm_windows_web_app.app
Removed azurerm_windows_web_app.app
Successfully removed 1 resource instance(s).
```

Figure 5.6: Output of the terraform state rm command

The command output confirms the resource deletion.

For more information about the `state rm` command, read the documentation here: `https://developer.hashicorp.com/terraform/cli/commands/state/rm`.

There's more...

One scenario in which I recently needed to manipulate Terraform state was when I updated the `azurerm` Terraform provider to version 3. I encountered some state schema errors during the execution of `terraform plan`.

To resolve these errors, I needed to delete the resources from state with the `terraform state rm` command and then import the resource into state under the new name (we will see that in detail in the *Importing existing resources* recipe in this chapter). To get more information about this scenario, read the documentation here: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/3.0-upgrade-guide#migrating-to-new--renamed-resources`.

In this recipe, we learned some commands to manipulate Terraform state. There are other interesting commands to use and to get the complete list of commands that manipulate state, read the documentation here: `https://developer.hashicorp.com/terraform/cli/state`.

In the next recipe, we will see another operation on Terraform state, which is the synchronization the Terraform state.

See also

- The learning lab about Terraform state management is available here: `https://developer.hashicorp.com/terraform/tutorials/cli/state-cli`.

Synchronizing Terraform state

In the previous recipes, we learned some operations to manage Terraform state to list, show, and remove resources from Terraform state.

There is another use case where resources already provisioned with Terraform configuration have been modified manually without using Terraform. The problem in this scenario is that the state file doesn't contain the configuration matching the configuration of the real provisioned resources.

The goal of this recipe is to learn how to synchronize or refresh the state file with the changes to the resources.

Let's get started!

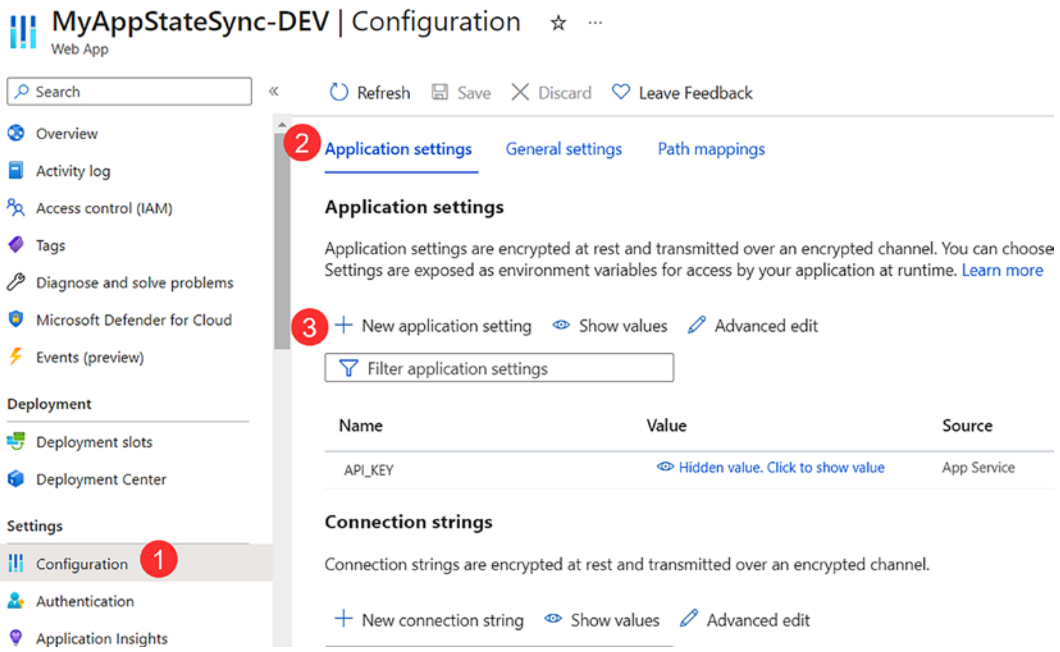
Getting ready

To get started with this recipe, we need to have already provisioned a basic Azure infrastructure composed of a resource group, a App Service plan, and an Azure Linux Web App.

The source code for the Terraform configuration of this infrastructure is here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/sync`.

After this deployment, we manually update one of the application settings directly in the Azure portal by navigating to the **Configuration** menu, then clicking on the **Application settings** tab, and finally clicking on the **New application setting** button.

The following screenshot shows the process to add the new application settings in Azure App Service:



Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose
Settings are exposed as environment variables for access by your application at runtime. Learn more

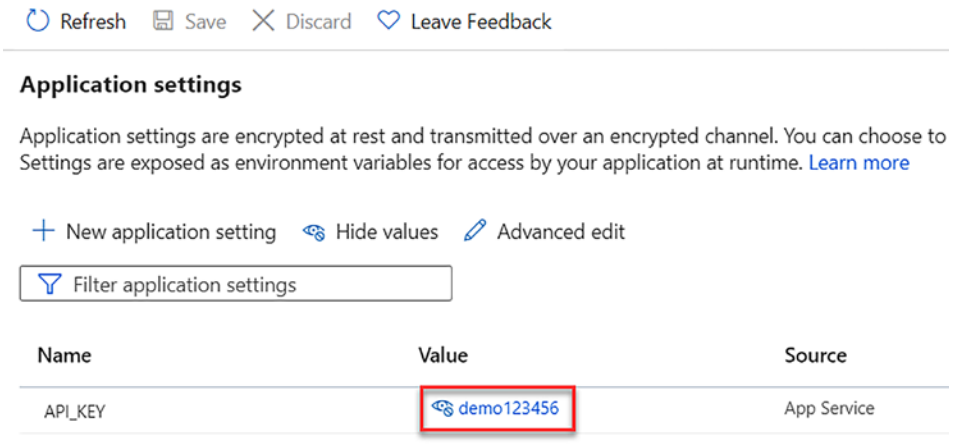
Name	Value	Source
API_KEY	Hidden value. Click to show value	App Service

Connection strings

Connection strings are encrypted at rest and transmitted over an encrypted channel.

Figure 5.7: Adding new Azure App Service app settings

The following screenshot shows the modified application settings in App Service:



Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose to
Settings are exposed as environment variables for access by your application at runtime. Learn more

Name	Value	Source
API_KEY	demo123456	App Service

Figure 5.8: Sample Azure App Service app settings

Because the next execution of the `terraform apply` command will replace this value with the original value, the goal of this recipe is to visualize the drift between Terraform state and the infrastructure, and then synchronize the state file with the real state of the provisioned infrastructure.

How to do it…

Perform the following steps to view the drift and synchronize Terraform state:

1. First, we want to view the difference between Terraform state and the real infrastructure; for this, we run the following command:

```
terraform plan -refresh-only
```

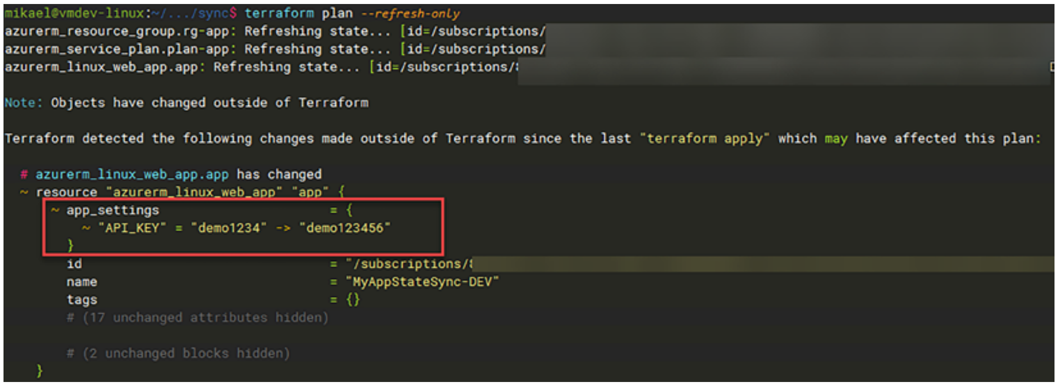
2. Then, to synchronize Terraform state according to the previous `plan` output, we will run this command:

   ```
   Terraform apply --refresh-only
   ```

3. Finally, we update the Terraform configuration to set the same value as the manually modified value.

How it works...

In *Step 1*, we run the command `terraform plan -refresh-only` to visualize the drift between the Terraform state file and the real infrastructure. The following image shows the output of this command:

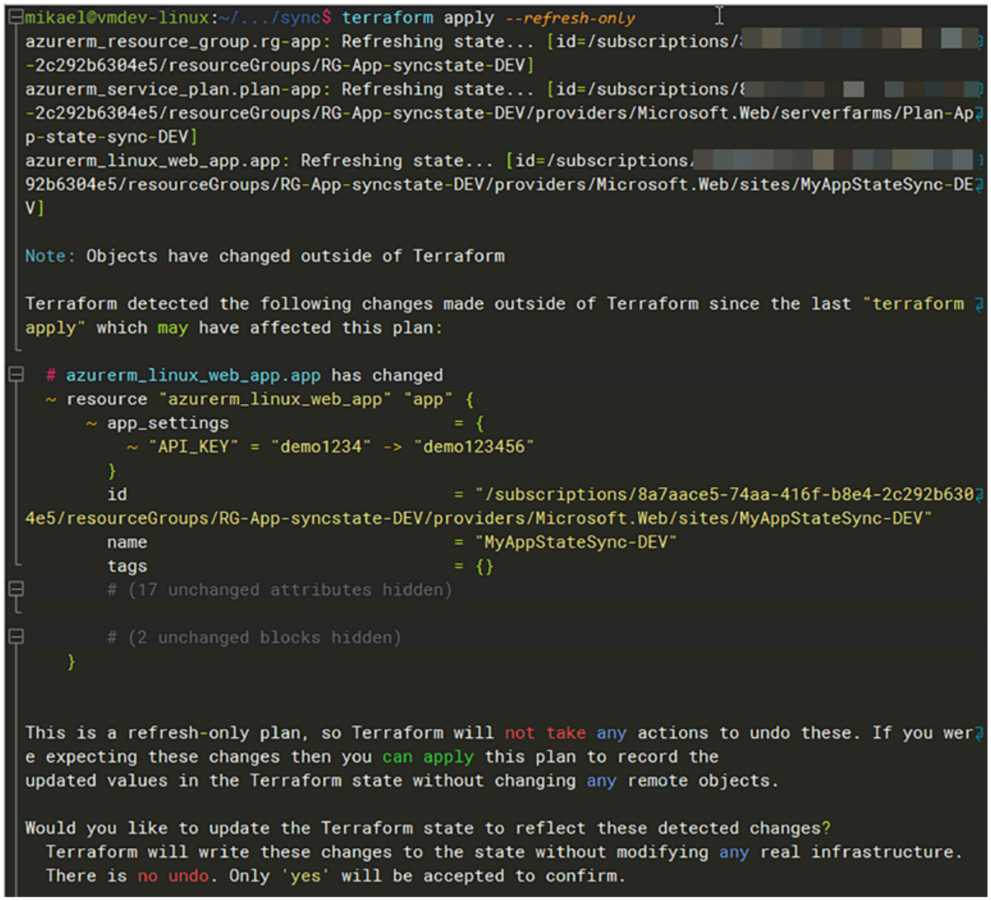


```
mikael@vmdev-linux:~/.../sync$ terraform plan --refresh-only
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/[redacted]
azurerm_service_plan.plan-app: Refreshing state... [id=/subscriptions/[redacted]
azurerm_linux_web_app.app: Refreshing state... [id=/subscriptions/[redacted]

Note: Objects have changed outside of Terraform

Terraform detected the following changes made outside of Terraform since the last "terraform apply" which may have affected this plan:

  # azurerm_linux_web_app.app has changed
  ~ resource "azurerm_linux_web_app" "app" {
      ~ app_settings                     = {
          ~ "API_KEY" = "demo1234" -> "demo123456"
        }
        id                               = "/subscriptions/[redacted]
        name                             = "MyAppStateSync-DEV"
        tags                             = {}
        # (17 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }
```

Figure 5.9: Terraform refresh during plan

We can see that Terraform detects that in Terraform state, the `API_KEY` settings value is `demo1234` and in the Web App, the value of this setting is `demo123456`.

In *Step 2*, we synchronize state by running the command `terraform apply --refresh-only`. The following image shows the output of this command:

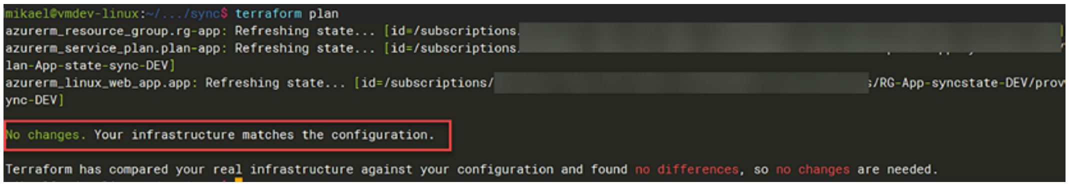


```
mikael@vmdev-linux:~/.../sync$ terraform apply --refresh-only
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/[redacted]
-2c292b6304e5/resourceGroups/RG-App-syncstate-DEV]
azurerm_service_plan.plan-app: Refreshing state... [id=/subscriptions/[redacted]
-2c292b6304e5/resourceGroups/RG-App-syncstate-DEV/providers/Microsoft.Web/serverfarms/Plan-Ap
p-state-sync-DEV]
azurerm_linux_web_app.app: Refreshing state... [id=/subscriptions/[redacted]
92b6304e5/resourceGroups/RG-App-syncstate-DEV/providers/Microsoft.Web/sites/MyAppStateSync-DE
V]

Note: Objects have changed outside of Terraform

Terraform detected the following changes made outside of Terraform since the last "terraform
apply" which may have affected this plan:

  # azurerm_linux_web_app.app has changed
  ~ resource "azurerm_linux_web_app" "app" {
      ~ app_settings                      = {
          ~ "API_KEY" = "demo1234" -> "demo123456"
        }
        id                                = "/subscriptions/8a7aace5-74aa-416f-b8e4-2c292b630
4e5/resourceGroups/RG-App-syncstate-DEV/providers/Microsoft.Web/sites/MyAppStateSync-DEV"
        name                              = "MyAppStateSync-DEV"
        tags                              = {}
        # (17 unchanged attributes hidden)

        # (2 unchanged blocks hidden)
    }


This is a refresh-only plan, so Terraform will not take any actions to undo these. If you wer
e expecting these changes then you can apply this plan to record the
updated values in the Terraform state without changing any remote objects.

Would you like to update the Terraform state to reflect these detected changes?
  Terraform will write these changes to the state without modifying any real infrastructure.
  There is no undo. Only 'yes' will be accepted to confirm.
```

Figure 5.10: Terraform refreshes during apply

We can see that this command only refreshes state and doesn't apply any changes to the resources.

Finally, we update the Terraform configuration with the same value as the API settings. If we run the `terraform plan` command, we get the following output:



```
mikael@vmdev-linux:~/.../sync$ terraform plan
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions.[redacted]
azurerm_service_plan.plan-app: Refreshing state... [id=/subscriptions.[redacted]
lan-App-state-sync-DEV]
azurerm_linux_web_app.app: Refreshing state... [id=/subscriptions/[redacted]s/RG-App-syncstate-DEV/prov
ync-DEV]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
```

Figure 5.11: No changes after refresh

We can see that there are no differences found. Therefore, there are no changes applied.

Now, the Terraform configuration, state, and the infrastructure have the same configuration.

There's more...

Before Terraform 0.15.4, we refreshed state by running the command `terraform refresh`, but this command has been deprecated, as mentioned here: `https://developer.hashicorp.com/terraform/cli/commands/refresh`.

See also

- The learning lab on the `refresh-only` of state is available here: `https://learn.hashicorp.com/tutorials/terraform/refresh?in=terraform/cli`
- The learning lab on drift detection is available here: `https://developer.hashicorp.com/terraform/tutorials/state/resource-drift`
- The documentation on the `refresh` command is available here: `https://developer.hashicorp.com/terraform/cli/commands/refresh`

Importing existing resources

So far in this book, we have seen the common use of Terraform, which is to write a Terraform configuration for infrastructure to be created by Terraform. This execution will provision or apply changes to an infrastructure, which will be reflected in the Terraform state file.

In the previous recipe, we learned to synchronize Terraform state to refresh it with updated resource properties.

In certain scenarios, however, it may be necessary to import entire resources that have already been provisioned the Terraform state. Examples of such scenarios include the following:

- Resources have been provisioned manually (or by scripts) and now it is desired that their configuration is in the Terraform configuration and in Terraform state.
- Terraform state that contains the configuration of an infrastructure has been corrupted or deleted and regeneration is desirable.

In this recipe, we will discuss how, with the assistance of Terraform commands, we can import the configuration of a resource that has already been provisioned inside Terraform state.

Getting ready

For this recipe, the requirement is to have an existing Terraform configuration: we will use the following Terraform configuration, which provisions an Azure Resource Group:

```
data azurerm_subscription "current" { }

resource "azurerm_resource_group" "rg-app" {
  name     = "RG-APP-IMPORT-${substr(data.azurerm_subscription.current.
subscription_id,0,5)}"
  location = "westeurope"
}
```

In the above Terraform configuration, we have added the first five numbers of the current subscription ID to the end of the resource group name. This allows us to make the name of our resource group unique.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/import`.

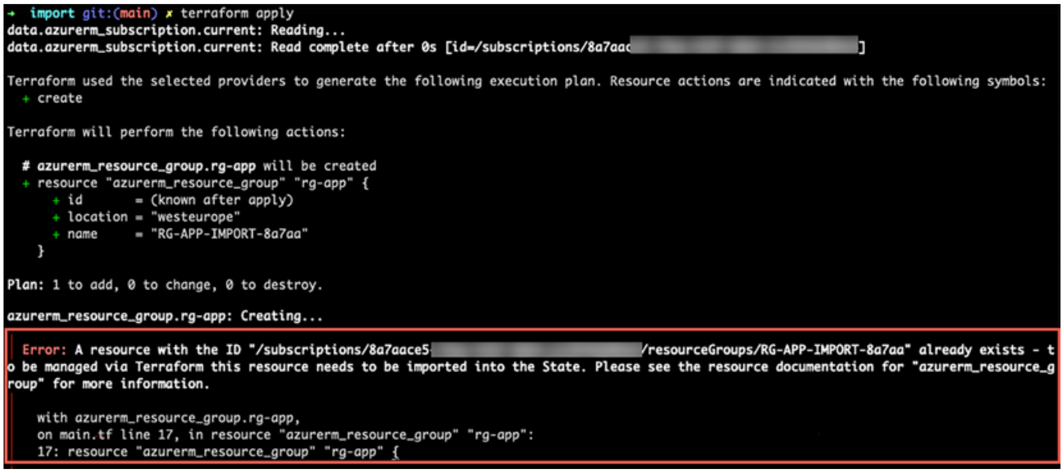
Also, in the Azure portal, we have created this resource group called RG-APP-IMPORT-<5 first number of subscription id> manually, as explained in the pertinent documentation: `https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal`.

The following screenshot shows this resource group in Azure:

RG-APP-IMPORT -8a7aa

Figure 5.12: Existing Azure resource group

At this point, if we run Terraform on this configuration, the `terraform apply` command will try to create this resource group. It will fail and return the error that the resource group already exists and cannot be created, as shown in the following screenshot:



```
→  import git:(main) ✗ terraform apply
data.azurerm_subscription.current: Reading...
data.azurerm_subscription.current: Read complete after 0s [id=/subscriptions/8a7aa[redacted]]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.rg-app will be created
  + resource "azurerm_resource_group" "rg-app" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "RG-APP-IMPORT-8a7aa"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

azurerm_resource_group.rg-app: Creating...

  Error: A resource with the ID "/subscriptions/8a7aace5[redacted]/resourceGroups/RG-APP-IMPORT-8a7aa" already exists - t
o be managed via Terraform this resource needs to be imported into the State. Please see the resource documentation for "azurerm_resource_g
roup" for more information.

    with azurerm_resource_group.rg-app,
    on main.tf line 17, in resource "azurerm_resource_group" "rg-app":
    17: resource "azurerm_resource_group" "rg-app" {
```

Figure 5.13: Terraform error – resource already exists

It is therefore necessary to use a resource import operation directly in Terraform state.

In this recipe, we will perform an import operation with one resource group in Azure. But it is important to note that each Terraform provider uses different formats of the ID for the `import` command, often based on the different needs of the underlying cloud APIs.

The goal of this recipe is to import the configuration of this resource group in Terraform state corresponding to our Terraform configuration.

How to do it...

Perform the following steps:

1. We initialize the Terraform context by executing the `init` command:

   ```
   terraform init
   ```

2. Then, we execute the terraform `import` command as follows:

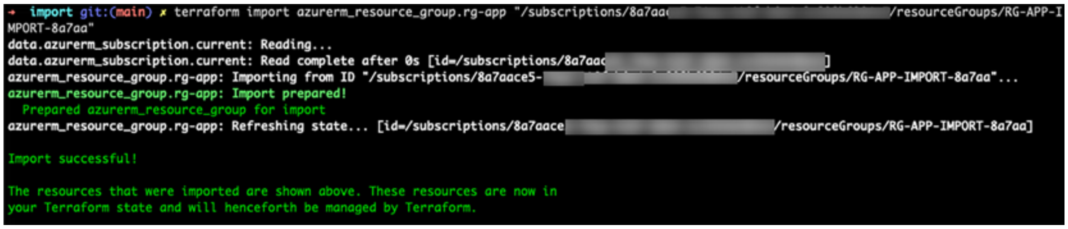
   ```
   terraform import azurerm_resource_group.rg-app "/subscriptions/<your
   azure subscriptionid>/resourceGroups/RG-APP-IMPORT-<5 first char of
   your subscription id>"
   ```

How it works...

In *Step 1*, the Terraform context is initialized with the `terraform init` command.

Then, in *Step 2*, we execute the `terraform import` command, which takes the reference name of the Terraform resource as the first parameter and the Azure identifier of the resource group as the second parameter.

The following screenshot shows the output of the execution of this command:



```
→  import git:(main) ✗ terraform import azurerm_resource_group.rg-app "/subscriptions/8a7aa[redacted]/resourceGroups/RG-APP-I
MPORT-8a7aa"
data.azurerm_subscription.current: Reading...
data.azurerm_subscription.current: Read complete after 0s [id=/subscriptions/8a7aa[redacted]]
azurerm_resource_group.rg-app: Importing from ID "/subscriptions/8a7aace5-[redacted]/resourceGroups/RG-APP-IMPORT-8a7aa"...
azurerm_resource_group.rg-app: Import prepared!
  Prepared azurerm_resource_group for import
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/8a7aace[redacted]/resourceGroups/RG-APP-IMPORT-8a7aa]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

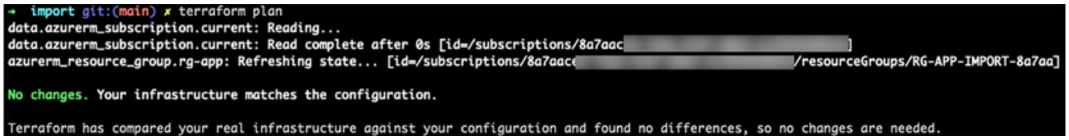
Figure 5.14: terraform import output

We can see that the resource was indeed imported into Terraform state.

We now have the Terraform configuration, the Terraform state file, and the resources in Azure up to date.

There's more...

To check the execution of the resource import, we execute the `terraform plan` command and it should be such that no changes are required, as can be seen in the following screenshot:



```
→  import git:(main) ✗ terraform plan
data.azurerm_subscription.current: Reading...
data.azurerm_subscription.current: Read complete after 0s [id=/subscriptions/8a7aac[redacted]]
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/8a7aac[redacted]/resourceGroups/RG-APP-IMPORT-8a7aa]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
```

Figure 5.15: terraform plan after import

We can see in the preceding output of the `terraform plan` command, that the resource has been imported into Terraform state and the change is applied. Additionally, since Terraform 1.5.0 was released, we can import resources by using the `import` block (the documentation is available at `https://developer.hashicorp.com/terraform/language/import`) in the Terraform configuration. For example, in our Terraform configuration studied in this recipe, in the `main.tf` file, we add the following Terraform configuration under the `azurerm_resource_group` resource:

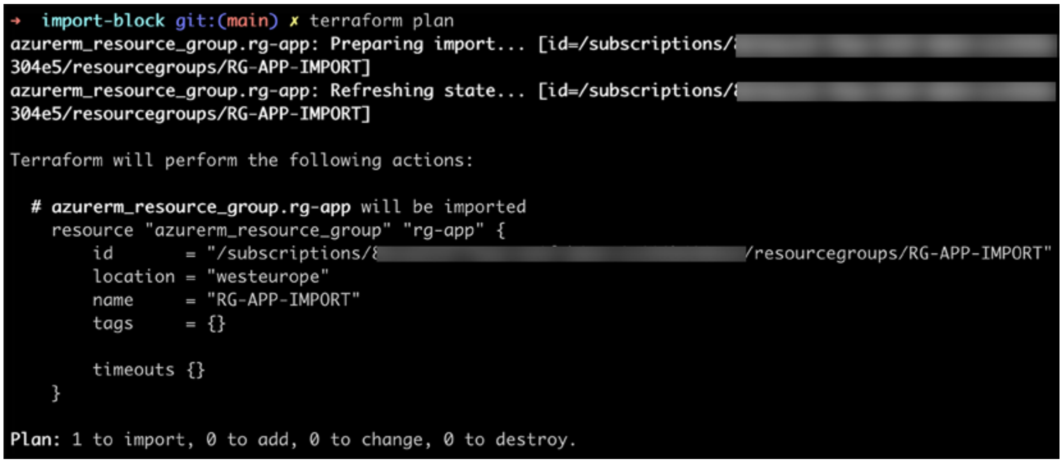
```
import {
  id = "/subscriptions/<your subscription id>/resourcegroups/RG-APP-
IMPORT-<5 first char of subcription id>"
  to = azurerm_resource_group.rg-app
}
```

In the preceding configuration:

- The `id` property is the ID of the resource (here, it is the Azure Group ID)
- The `to` property is the name of the Terraform resource (here, it is `azurerm_resource_group.rg-app`)

The source code of the preceding Terraform configuration is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/import-block`. Remember to update the code with your subscription ID.

Then run the `terraform init`, `plan`, and `apply` commands to perform the import operation. The following screenshot shows the execution of the `terraform plan` command:



```
→  import-block git:(main) ✗ terraform plan
azurerm_resource_group.rg-app: Preparing import... [id=/subscriptions/[redacted]
304e5/resourcegroups/RG-APP-IMPORT]
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/[redacted]
304e5/resourcegroups/RG-APP-IMPORT]

Terraform will perform the following actions:

  # azurerm_resource_group.rg-app will be imported
    resource "azurerm_resource_group" "rg-app" {
        id       = "/subscriptions/[redacted]/resourcegroups/RG-APP-IMPORT"
        location = "westeurope"
        name     = "RG-APP-IMPORT"
        tags     = {}

        timeouts {}
    }

Plan: 1 to import, 0 to add, 0 to change, 0 to destroy.
```

Figure 5.16: terraform plan with import block

We can see that Terraform will perform one import operation with the details of the imported resource.

After the execution of the `terraform apply` command, Terraform state will be updated with the Azure resource group configuration. What's more, with this new import functionality described directly in the Terraform configuration, we can go even further by generating the Terraform configuration of the resource we want to import.

For this generation, let's suppose we have a `main.tf` file that has no resources, in which we have only the following provider configuration:

```
terraform {
  required_version = "~> 1.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.20"
    }
  }
}

provider "azurerm" {
  features {}
}
```

And only the `import` block as shown here:

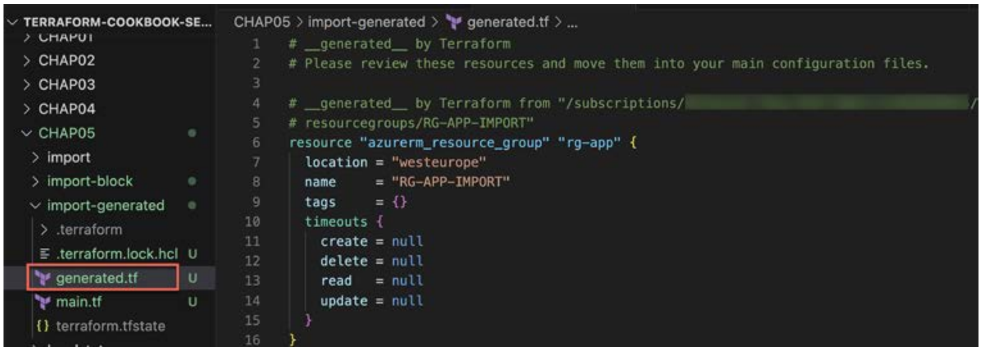
```
import {
  id = "/subscriptions/<your subscription id>/resourcegroups/RG-APP-
IMPORT"
  to = azurerm_resource_group.rg-app
}
```

The source code for the Terraform configuration is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/import-generated`. Remember to update the code with your subscription ID.

To generate the Terraform configuration for the Azure resource group `RG-APP-IMPORT` and import it in Terraform state, add the option `--generate-config-out` in the `terraform plan` command. To complete the import process (generating the configuration and importing into state) run the following commands:

1. Run `terraform init`.
2. Run `terraform plan --generate-config-out=generated.tf`.

3. At this step, the `plan` command execution will generate a new file, `generated.tf,` that contains the Terraform configuration of the resource group. The following screenshot shows the generated Terraform configuration:



```
# __generated__ by Terraform
# Please review these resources and move them into your main configuration files.

# __generated__ by Terraform from "/subscriptions/[redacted]/
# resourcegroups/RG-APP-IMPORT"
resource "azurerm_resource_group" "rg-app" {
  location = "westeurope"
  name     = "RG-APP-IMPORT"
  tags     = {}
  timeouts {
    create = null
    delete = null
    read   = null
    update = null
  }
}
```

Figure 5.17: Generated Terraform configuration using import block

- Finally, run the `terraform apply` command to perform the import in Terraform state.

> Note that if you need to generate Terraform configuration for Azure resources you can use the Azure specific tool `aztfexport` as explained in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*, in the *Generating a Terraform configuration for existing Azure infrastructure recipe*.

See also

- The documentation for the `import` command is available here: `https://www.terraform.io/docs/commands/import.html`.
- The documentation of the `import` block is available here: `https://developer.hashicorp.com/terraform/language/import`

Using external resources from other Terraform state files

In the *Obtaining external data with data sources* recipe in *Chapter 4*, *Using Terraform with External Data*, we saw that it's possible to retrieve information about resources already present in the infrastructure using data blocks.

In this recipe, we will learn that it is also possible to retrieve external information that is present in other Terraform state files.

Getting ready

For this recipe, we will, like the previous recipe, use a Terraform configuration that provisions an Azure App Service, which must be part of an already provisioned an App Service plan.

Unlike the previous recipe, we will not use individual data sources; instead, we will read outputs from an existing Terraform state that was used to provision the App Service plan.

As a prerequisite, in the Terraform configuration that was used to provision the App Service plan, we must have an output value (read the *Using outputs to expose Terraform provisioned data* recipe in *Chapter 2*, *Writing Terraform Configurations*, to learn how to use an output value) that returns the identifier of the App Service plan, as shown in the following configuration:

```
terraform {
  …
  backend "azurerm" {
   resource_group_name = "rg_tfstate"
   storage_account_name = "storstate"
   container_name = "tfbackends"
   key = "serviceplan.tfstate"
  }

}

resource "azurerm_service_plan" "plan-app" {
  name = "MyServicePlan"
  location = "westeurope"
  resource_group_name = "myrg"
  …
}

output "service_plan_id" {
  description = "The service plan"
  value = azurerm_service_plan.plan-app.id
}
```

To provision all resources (of this remote backend), you can use these scripts that create and destroy these resources here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/remote-backend`.

In addition, we used a remote backend version of Azure Storage (see the *Protecting state files in an Azure remote backend* recipe in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*, for more information) to store Terraform state of the App Service plan.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/remote-state`.

How to do it…

Perform the following steps:

1. In the Terraform configuration that provides the Azure App Service, add and configure the `terraform_remote_state` block, as follows:

```
data "terraform_remote_state" "service_plan_tfstate" {
  backend = "azurerm"
  config = {
     resource_group_name = "rg_tfstate"
     storage_account_name = "storstate"
     container_name = "tfbackends"
     key = "serviceplan.tfstate"
  }
}
```

2. Then, in the Terraform configuration of the Azure App Service, use the created output of the App Service plan, as follows:

```
resource "azurerm_windows_web_app" "app" {
  name = "${var.app_name}-${var.environement}"
  location = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id = data.terraform_remote_state.service_plan_
tfstate.outputs.service_plan_id
  site_config {}
}
```

How it works…

In *Step 1*, we added the `terraform_remote_state` block, which allows us to retrieve outputs present in another Terraform state file. In its block, we specified the remote backend information, which is where the given Terraform state is stored (in this recipe, we used Azure Storage).

In *Step 2*, we used the ID returned by the output present in the Terraform state file.

The result of executing this configuration is the same as what we learned in the *Using external resources with data sources* recipe in *Chapter 4*, *Using Terraform with External Data*.

There's more...

This technique is very practical when separating a Terraform configuration that deploys a complex infrastructure.

Separating the Terraform configuration is a good practice because it allows better control and maintainability of the Terraform configuration. It also allows us to provision each part separately, without it impacting the rest of the infrastructure.

To know when to use a data block or a `terraform_remote_state` block, the following recommendations must be kept in mind:

The (Azure-specific) `data` source is used in the following cases:

- When external resources have not been provisioned with Terraform configuration (it has been built manually or with a script)
- When the user providing the resources of our Terraform configuration does not have access to the remote backend

The `terraform_remote_state` data source is used in the following cases:

- When external resources have been provisioned with Terraform configuration
- When the user providing the resources of our Terraform configuration has read access to the other remote backend
- When the external Terraform state file contains the output of the property that we need in our Terraform configuration

See also

- The documentation for the `terraform_remote_state` block is available at `https://www.terraform.io/docs/providers/terraform/d/remote_state.html`.

Refactoring resources in configuration

In the previous recipes, we learned how to manage resources in Terraform state using the Terraform CLI.

In some situations, we may want to refactor the Terraform configuration by:

- Renaming resources or modules to be more consistent with the resource role
- Adding `count` or `for_each` iterators inside existing resources to scale the provisioning of the resource from 1 instance to *N* instances
- Moving resources inside a module to refactor Terraform configuration

What you need to know is that any change in the Terraform configuration brings about changes in the Terraform state file.

In this recipe, we will learn how to perform a simple and basic refactoring operation so as not to destroy the resources that already exist in state, using two methods:

- Using the Terraform CLI
- Using the `moved` block

Let's get started!

Getting ready

To complete this recipe, we will start with a simple Terraform configuration that creates two Azure Linux Web Apps (inside the same App Service plan) using the following configuration (called script 1):

```
resource "azurerm_virtual_network" "vnet" {
  name                = "vnet1"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
}

resource "azurerm_subnet" "snet1" {
  name                 = "subnet1"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}

resource "azurerm_subnet" "snet2" {
  name                 = "subnet2"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.2.0/24"]
}
```

We run the `terraform init`, `plan`, and `apply` commands.

After reflection, we want to refactor the configuration to use only one resource, `azurerm_subnet`, and use the `foreach` meta-argument to loop over a list.

In this Terraform configuration, we will directly replace the code with the configuration that we want with the following code (called script 2):

```
locals {
  subnet_list = {
    subnet1 = "10.0.1.0/24"
    subnet2 = "10.0.2.0/24"
  }
}



resource "azurerm_subnet" "snetlist" {
  for_each             = local.subnet_list
  name                 = each.key
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = [each.value]
}
```

At the `terraform apply` execution, Terraform will delete the first two `azurerm_subnet` resources that are in the first code, and recreate two new `azurerm_subnet` resources.

But the goal of the refactorization isn't to delete and recreate new resources. The goal is just to refactor the Terraform configuration and change Terraform state accordingly so no changes need to be applied when `terraform apply` is executed.

The goal of this recipe is to refactor this configuration by using the Terraform CLI first, and then using the `moved` block.

The complete source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/refactor`.

How to do it...

To refactor our initial Terraform configuration (script 1), using the Terraform CLI, perform the following steps:

In the `cli` subfolder, run the `terraform init`, `plan`, and `apply` commands

Use the command `terraform state mv` to move each resource in Terraform state by running the following commands:

```
terraform state mv 'azurerm_subnet.snet1' 'azurerm_subnet.
snetlist["subnet1"]'
terraform state mv 'azurerm_subnet.snet2' 'azurerm_subnet.
snetlist["subnet2"]'
```

On Windows, we can avoid using double quotes by instead using a backslash, for example: `terraform state mv 'azurerm_subnet.snet1' 'azurerm_subnet.snetlist[\"subnet1\"]'`.

The following article explains this further: `https://devcoops.com/terraform-powershell-escape-double-quotes/`

Then, we can write the refactorized configuration (script 2) and delete the old configuration (script 1).

Finally, we run the `plan` command.

The source code for this configuration is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/refactor/cli`.

To refactor our initial Terraform configuration (script 1), using the Terraform `moved` block, perform the following steps:

1. In the Terraform configuration, write the final refactorized configuration (script 2).
2. Then, in the same Terraform configuration file, add the following code:

```
moved {
  from = azurerm_subnet.snet1
  to   = azurerm_subnet.snetlist["subnet1"]
}
```

```
moved {
  from = azurerm_subnet.snet2
  to   = azurerm_subnet.snetlist["subnet2"]
}
```

3. Delete the old Terraform configuration (script 1).

Finally, we run the `terraform init` and `apply` commands.

Remove the two `moved` blocks written in *Step 2* from the configuration.

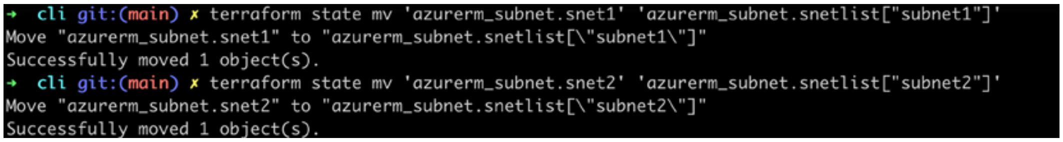
The source code for this configuration is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/refactor/moved`.

How it works...

In the first part of this recipe, in the first solution, we run the `terraform state mv` command, which moves a resource inside another resource, like a list or module, and we can also use this command to rename a Terraform resource.

We run this command for each `azurerm_subnet` (`snet1` and `snet2`) to move it inside the new `azurerm_subnet` resource snetlist.

The following image shows the execution of this command:



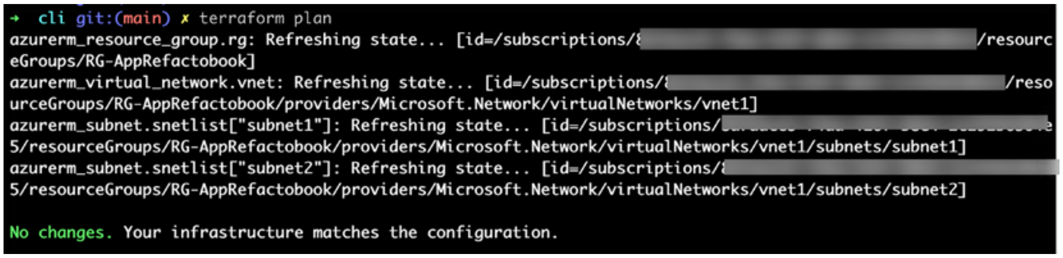
```
➜  cli git:(main) ✗ terraform state mv 'azurerm_subnet.snet1' 'azurerm_subnet.snetlist["subnet1"]'
Move "azurerm_subnet.snet1" to "azurerm_subnet.snetlist[\"subnet1\"]"
Successfully moved 1 object(s).
➜  cli git:(main) ✗ terraform state mv 'azurerm_subnet.snet2' 'azurerm_subnet.snetlist["subnet2"]'
Move "azurerm_subnet.snet2" to "azurerm_subnet.snetlist[\"subnet2\"]"
Successfully moved 1 object(s).
```

Figure 5.18: terraform state mv output

We can see the output of this command, which indicates the `move` operation has been completed successfully.

After this operation, we can now refactor the code and write the final configuration, which includes a `locals` expression with a list of random string lengths and the `count` expression, and we remove the old configuration.

Finally, we run the Terraform workflow with the `init` and `plan` commands. The following image shows the result of the `plan` command:



```
→  cli git:(main) ✗ terraform plan
azurerm_resource_group.rg: Refreshing state... [id=/subscriptions/[redacted]/resourc
eGroups/RG-AppRefactobook]
azurerm_virtual_network.vnet: Refreshing state... [id=/subscriptions/[redacted]/reso
urceGroups/RG-AppRefactobook/providers/Microsoft.Network/virtualNetworks/vnet1]
azurerm_subnet.snetlist["subnet1"]: Refreshing state... [id=/subscriptions/[redacted]
5/resourceGroups/RG-AppRefactobook/providers/Microsoft.Network/virtualNetworks/vnet1/subnets/subnet1]
azurerm_subnet.snetlist["subnet2"]: Refreshing state... [id=/subscriptions/[redacted]
5/resourceGroups/RG-AppRefactobook/providers/Microsoft.Network/virtualNetworks/vnet1/subnets/subnet2]

No changes. Your infrastructure matches the configuration.
```

Figure 5.19: terraform plan after terraform state mv

We can see that there are no changes to be applied and our random string value is kept intact inside Terraform state.

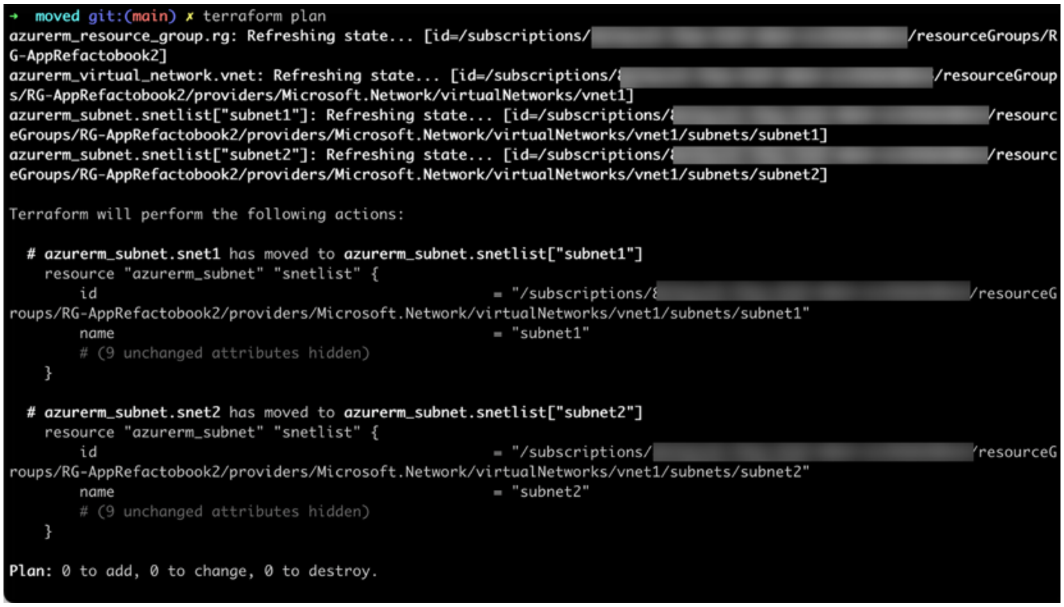
In the second solution, we use the `moved` block, which was introduced in Terraform 1.1.

To apply this solution, we add two `moved` blocks in our initial configuration, which contain two properties: `from`, which is the Terraform initial resource name, and `to`, which is the name of the target resource name. For example, for the first `azurerm_subnet`, we write the following code:

```
moved {
  from = azurerm_subnet.snet1
  to   = azurerm_subnet.snetlist["subnet1"]
}
```

Then, we delete the initial configuration and finally we run the Terraform workflow with `init`, `plan`, and `apply` commands.

The following image shows the execution of the `plan` command:



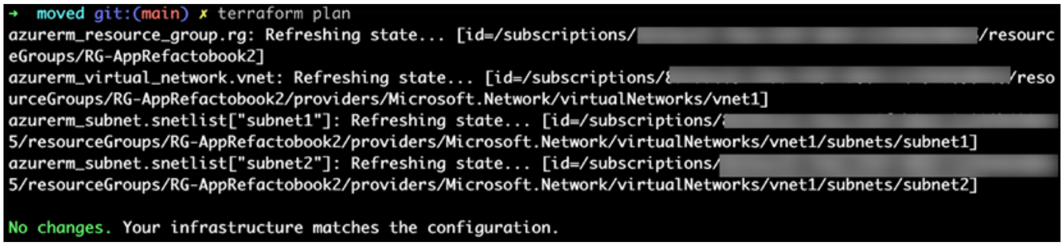
```
→ moved git:(main) ✗ terraform plan
azurerm_resource_group.rg: Refreshing state... [id=/subscriptions/[redacted]/resourceGroups/R
G-AppRefactobook2]
azurerm_virtual_network.vnet: Refreshing state... [id=/subscriptions/[redacted]/resourceGroup
s/RG-AppRefactobook2/providers/Microsoft.Network/virtualNetworks/vnet1]
azurerm_subnet.snetlist["subnet1"]: Refreshing state... [id=/subscriptions/[redacted]/resourc
eGroups/RG-AppRefactobook2/providers/Microsoft.Network/virtualNetworks/vnet1/subnets/subnet1]
azurerm_subnet.snetlist["subnet2"]: Refreshing state... [id=/subscriptions/[redacted]/resourc
eGroups/RG-AppRefactobook2/providers/Microsoft.Network/virtualNetworks/vnet1/subnets/subnet2]

Terraform will perform the following actions:

  # azurerm_subnet.snet1 has moved to azurerm_subnet.snetlist["subnet1"]
    resource "azurerm_subnet" "snetlist" {
        id                                             = "/subscriptions/[redacted]/resourceG
roups/RG-AppRefactobook2/providers/Microsoft.Network/virtualNetworks/vnet1/subnets/subnet1"
        name                                           = "subnet1"
        # (9 unchanged attributes hidden)
    }

  # azurerm_subnet.snet2 has moved to azurerm_subnet.snetlist["subnet2"]
    resource "azurerm_subnet" "snetlist" {
        id                                             = "/subscriptions/[redacted]/resourceG
roups/RG-AppRefactobook2/providers/Microsoft.Network/virtualNetworks/vnet1/subnets/subnet2"
        name                                           = "subnet2"
        # (9 unchanged attributes hidden)
    }

Plan: 0 to add, 0 to change, 0 to destroy.
```

Figure 5.20: Terraform moved in plan

We can see that Terraform will move the desired resources and will not make any changes to the value of the resources.

To apply this `move` operation, we run the `terraform apply` command.

After the `apply` command, we can re-run `terraform plan` and see that there are no changes, as in the following image:



```
→ moved git:(main) ✗ terraform plan
azurerm_resource_group.rg: Refreshing state... [id=/subscriptions/[redacted]/resourc
eGroups/RG-AppRefactobook2]
azurerm_virtual_network.vnet: Refreshing state... [id=/subscriptions/[redacted]/reso
urceGroups/RG-AppRefactobook2/providers/Microsoft.Network/virtualNetworks/vnet1]
azurerm_subnet.snetlist["subnet1"]: Refreshing state... [id=/subscriptions/[redacted]
5/resourceGroups/RG-AppRefactobook2/providers/Microsoft.Network/virtualNetworks/vnet1/subnets/subnet1]
azurerm_subnet.snetlist["subnet2"]: Refreshing state... [id=/subscriptions/[redacted]
5/resourceGroups/RG-AppRefactobook2/providers/Microsoft.Network/virtualNetworks/vnet1/subnets/subnet2]

No changes. Your infrastructure matches the configuration.
```

Figure 5.21: terraform plan after apply using the moved block

To finish the refactoring process, we remove the configuration of the 2 `moved` blocks.

There's more...

In this recipe, we learned two ways to refactor Terraform configuration; the first way is by using the `terraform state mv` command and the second way is by using the `moved` block, which we write in the Terraform configuration.

You are probably wondering which solution to choose to refactor your code.

To help you, here is a list of the pros and cons of each of these solutions.

Using the `terraform state mv` command:

Pros:

- We can use this command to move a resource from one state to another. For more details, read this documentation: `https://developer.hashicorp.com/terraform/tutorials/state/state-cli#move-a-resource-to-a-different-state-file`.
- This can be integrated into an existing migration script if we need to refactor a lot of configurations.

Cons:

- This command runs the move directly without asking the user for confirmation.
- If we use a CI/CD pipeline, we need to add this command inside the external script and ensure that it runs.

Using the Terraform `moved` block:

Pros:

- We write the operation of the refactor inside the Terraform configuration.
- The move operation is integrated into the Terraform workflow. With `plan` and `apply`, we can decide to not apply the move if the result of the `plan` command isn't what we want to move.

Cons:

- If the Terraform workflow is executed inside an automated pipeline and the code is sourced to SVC (like Git), we need to perform multiple operations of commit and push to operate the refactoring with the following workflow:
    1. Add the `moved` block and new Terraform configuration.
    2. Commit and push.

3. The pipeline runs the Terraform workflow.
4. Remove the `moved` blocks.
5. Commit and push the clean and refactorized code.

- As the `moved` block is in a specific Terraform configuration, we can't use it to move a resource to another state.
- The `moved` block solves a common pain point by providing a safe migration path for `external` modules, from the module author's perspective. For example, imagine that as a module author, you wish to change the structure, but do it without impacting consumers of that module. Prior to introducing the `moved` blocks, most authors would rather avoid refactoring, or communicate it very carefully through major version changes and changelogs – e.g., v1 -> v2. On a similar note, the recommendations about removal are different in this context since the author cannot assume a particular upgrade has already happened (since they're not in control of state, consumers are). They'd typically leave the "`moved{}`" blocks in place for much longer, perhaps throughout the whole major release.

My personal advice is there isn't a perfect solution, so choose the method that is adapted to your scenario and your context.

Finally, in this recipe, we learned about just one scenario for Terraform configuration refactoring. For more use cases, read the documentation here: `https://developer.hashicorp.com/terraform/language/modules/develop/refactoring`.

See also

- The documentation for the `terraform state mv` command is available here: `https://developer.hashicorp.com/terraform/cli/commands/state/mv`
- The documentation for the Terraform `moved` block is available here: `https://developer.hashicorp.com/terraform/language/modules/develop/refactoring`
- A tutorial about refactoring with a `moved` block is available here: `https://developer.hashicorp.com/terraform/tutorials/configuration-language/move-config`
- A video about the use of the `moved` block is available here: `https://nedinthecloud.com/2021/12/14/using-the-moved-block-in-terraform-1-1/`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

6

Applying a Basic Terraform Workflow

Terraform is an **Infrastructure as Code** (**IaC**) tool that consists of linked elements: the Terraform configuration, written in **HashiCorp Configuration Language** (**HCL**), which describes the infrastructure we want to provision; the Terraform CLI, which will analyze and execute our Terraform configuration; and the Terraform State. In *Chapter 2*, *Writing Terraform Configurations*, and *Chapter 3*, *Scaling Your Infrastructure with Terraform*, we studied a variety of recipes on writing Terraform configuration using variables, loops, functions, and expressions of the language.

In this chapter, we will focus on the use of the Terraform CLI with its commands and options to operate the basic Terraform workflow. We will discuss how to present the configuration well and validate the syntax, the destruction of resources, how to list used providers, and the use of workspaces. Then we will learn the taint functionality, and we will see how to generate a dependency graph. Finally, we will see how to evaluate Terraform expressions and debug the execution of Terraform.

We will cover the following recipes in this chapter:

- Keeping your Terraform configuration clean
- Validating the code syntax
- Destroying infrastructure resources
- Displaying a list of providers used in a configuration
- Generating one Terraform lock file with Windows and Linux compatibility
- Copying a Terraform module

- Using workspaces to manage environments
- Exporting the output in JSON
- Tainting resources
- Generating the dependency graph
- Using different Terraform configuration directories
- Testing and evaluating Terraform expressions
- Debugging the Terraform execution

Technical requirements

Contrary to the previous chapters, the code examples provided in this chapter are not fundamental, since we will focus on the execution of Terraform command lines.

In this chapter, to provide Terraform configuration samples, we will manage resources in the Azure cloud; it is obvious that this also applies to all other Terraform providers. If you want to apply these recipes and don't have an Azure account, you can create an Azure account for free at this site: `https://azure.microsoft.com/en-us/free/`.

Additionally, to execute Terraform commands with the CLI, we use a command-line terminal (CMD, PowerShell, Bash, and so on), and the execution folder will be the folder containing the Terraform configuration of the recipe. This will apply to all recipes in this chapter.

The code examples in this chapter are available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06`.

Keeping your Terraform configuration clean

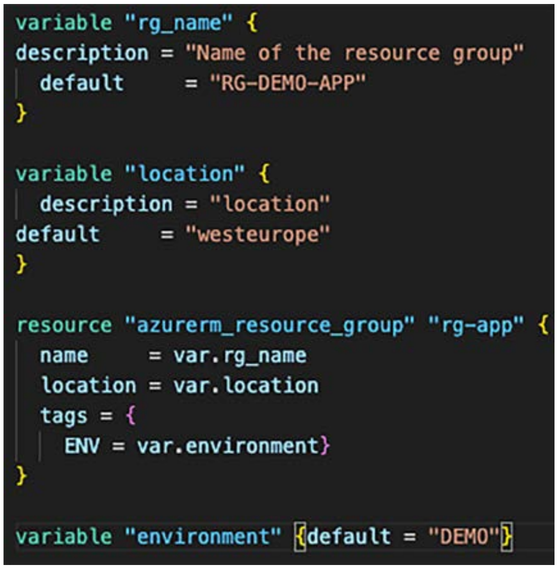
In any application with code, it is very important that the code is clean and clearly readable by all contributors (current and future) who will be involved in its maintenance and evolution.

In IaC and with Terraform, it is even more important to have clean code because written code serving as documentation is an advantage of IaC.

In this recipe, we will look at how to use Terraform's command line to properly format its code, and we will also see some tips to automate it.

Getting ready

To get started, we will start with a `main.tf` file that contains the following Terraform configuration:



```
variable "rg_name" {
description = "Name of the resource group"
  default     = "RG-DEMO-APP"
}

variable "location" {
  description = "location"
default     = "westeurope"
}

resource "azurerm_resource_group" "rg-app" {
  name     = var.rg_name
  location = var.location
  tags = {
    ENV = var.environment}
}

variable "environment" {default = "DEMO"}
```

Figure 6.1: A bad Terraform configuration format style

As we can see, this code is not very readable; it needs to be better formatted.

How to do it…

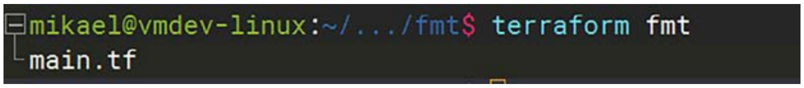
To fix the code indentation, execute the `terraform fmt` command at the root Terraform configuration, as follows:

```
terraform fmt
```

How it works…

The `terraform fmt` command makes it easier to arrange the code with the correct indentation.

At the end of its execution, this command displays the list of files that have been modified, as shown in the following screenshot:



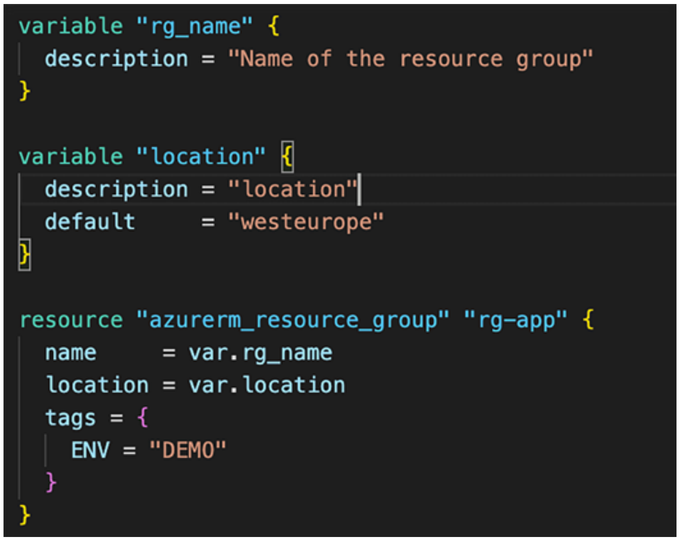
```
mikael@vmdev-linux:~/.../fmt$ terraform fmt
main.tf
```

Figure 6.2: The Terraform fmt command execution

The output of this command displays all fixed and formatted Terraform files.

We can see in the Terraform configuration that executing the `terraform fmt` command modified our `main.tf` file.

Then, we open the `main.tf` file and read it:



```
variable "rg_name" {
  description = "Name of the resource group"
}

variable "location" {
  description = "location"
  default     = "westeurope"
}

resource "azurerm_resource_group" "rg-app" {
  name     = var.rg_name
  location = var.location
  tags = {
    ENV = "DEMO"
  }
}
```

Figure 6.3: The well-formatted Terraform configuration

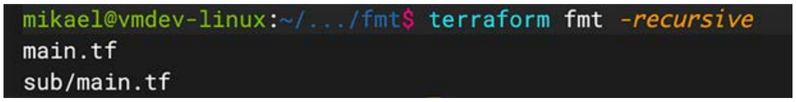
We can see in the preceding screenshot that the code has been well indented and so is more easily readable.

There's more...

In this recipe, we learned about the `terraform fmt` command executed in its most basic way – that is, without any additional options.

This default command indents any Terraform files, which are at the root of the current folder. We can also execute this command recursively – that is, it can also indent the code in subfolders of the current folder.

To do this, we execute the `terraform fmt` command with the `-recursive` option. The output of this is shown in the following screenshot:



```
mikael@vmdev-linux:~/.../fmt$ terraform fmt -recursive
main.tf
sub/main.tf
```

Figure 6.4: The terraform fmt –recursive command execution

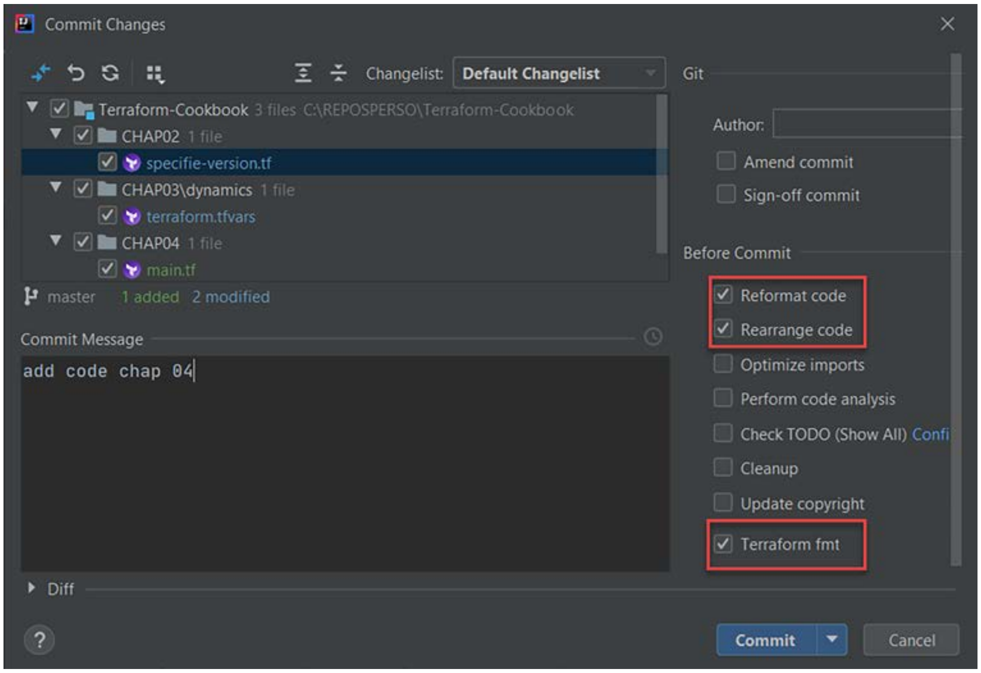
We see that the command has also formatted the `main.tf` file in the `sub` folder.

Among the other options of this command, there is also the `-check` option, which can be added and allows you to preview the files that will be indented, without applying the changes in the file(s).

Finally, it's also possible to automate the execution of this command, because apart from running it manually in a command terminal, as seen in this recipe, we can automate it to ensure that every time we save or commit a file in Git, the code provided and shared with the rest of the contributors will always be properly indented.

Thus, the IDEs that support Terraform have integrated the execution of this command natively with the writing of the code:

- With the Terraform extension of Visual Studio Code, we can have every Terraform file saved and formatted with the `terraform fmt` command. For more information, read the pertinent documentation: `https://marketplace.visualstudio.com/items?itemName=HashiCorp.terraform`.
- In IntelliJ IDEA, the **Save** action plugin enables the code to be formatted every time it is saved, and the **Terraform** plugin has a large integration of the `terraform fmt` command within the IDE. Furthermore, with this Terraform plugin, it is possible to execute the `terraform fmt` command and arrange the code at every code commit, as shown in the following screenshot:



- ✓ Terraform-Cookbook 3 files C:\REPOSPERSO\Terraform-Cookbook
  - ✓ CHAP02 1 file
    - ✓ specifie-version.tf
  - ✓ CHAP03\dynamics 1 file
    - ✓ terraform.tfvars
  - ✓ CHAP04 1 file
    - ✓ main.tf

master 1 added 2 modified

Commit Message

```
add code chap 04
```

Author:

- ☐ Amend commit
- ☐ Sign-off commit

Before Commit

- ✓ Reformat code
- ✓ Rearrange code
- ☐ Optimize imports
- ☐ Perform code analysis
- ☐ Check TODO (Show All) Confi
- ☐ Cleanup
- ☐ Update copyright
- ✓ Terraform fmt

Figure 6.5: The IntelliJ Terraform plugin

For more information on the Save action plugin, refer to `https://plugins.jetbrains.com/plugin/7642-save-actions`, and for the Terraform plugin, refer to `https://plugins.jetbrains.com/plugin/7808-hashicorp-terraform--hcl-language-support`.

For Git commits, it's possible to automate the execution of the `terraform fmt` command before each commit by using `pre-commits` that are hooks in Git: `https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`.

We will learn about this pre-commit hook in *Chapter 12*, *Deep-Diving into Terraform*, in the recipe *Checking configuration before commit using Git hooks pre-commit recipe*, when we learn how to automate the `fmt` command in a Git `pre-commit` hook.

See also

The complete `terraform fmt` command documentation is available here: `https://www.terraform.io/docs/commands/fmt.html`.

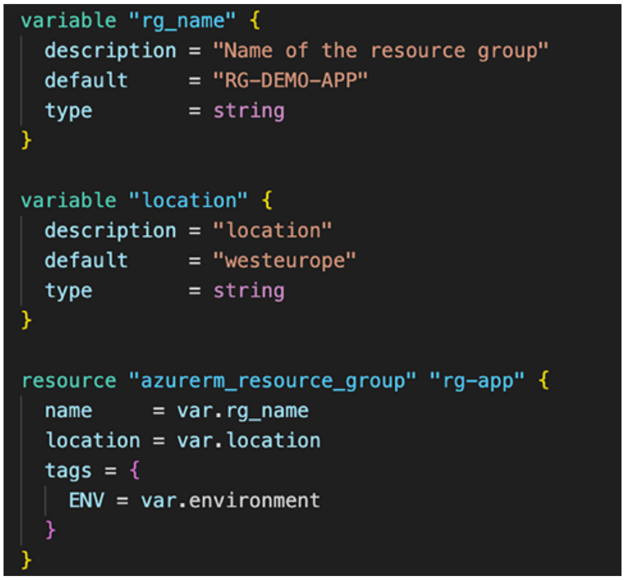
Validating the code syntax

When writing a Terraform configuration, it is important to be able to validate the syntax of the configuration we are writing before executing it, or even before archiving it in a Git repository.

We will see in this recipe how, by using the Terraform binary, we can check the validity of Terraform configuration syntax.

Getting ready

For this recipe, we will start with the following Terraform configuration, which is written in a `main.tf` file:



```
variable "rg_name" {
  description = "Name of the resource group"
  default     = "RG-DEMO-APP"
  type        = string
}

variable "location" {
  description = "location"
  default     = "westeurope"
  type        = string
}

resource "azurerm_resource_group" "rg-app" {
  name     = var.rg_name
  location = var.location
  tags = {
    ENV = var.environment
  }
}
```

Figure 6.6: Bad Terraform configuration

What we notice in the preceding code is that the declaration of the `environment` variable is missing.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/validate`.

How to do it...

To validate our Terraform configuration syntax, perform the following steps:

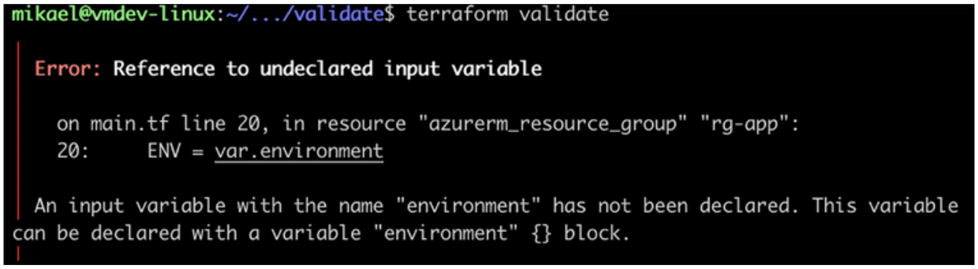
1. To start, initialize the Terraform context by running the following command:

   ```
   terraform init
   ```

2. Then, validate the code by executing the `validate` command:

   ```
   terraform validate
   ```

At the end of the execution of this command, we get the following output:

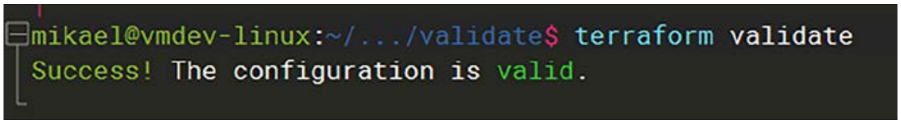


```
mikael@vmdev-linux:~/.../validate$ terraform validate

 Error: Reference to undeclared input variable

   on main.tf line 20, in resource "azurerm_resource_group" "rg-app":
   20:     ENV = var.environment

 An input variable with the name "environment" has not been declared. This variable
can be declared with a variable "environment" {} block.
```

Figure 6.7: The terraform validate command execution with an output error

3. We can see that there is one syntax error in the Terraform configuration, which indicates that we call the variable `var.environment`, which has not been declared.

   So, we correct the code and run the `terraform validate` command again until we have no more errors, as shown in the following screenshot:



```
mikael@vmdev-linux:~/.../validate$ terraform validate
Success! The configuration is valid.
```

Figure 6.8: The Terraform configuration is valid

The output shows us that the Terraform configuration is valid.

How it works...

In *Step 1*, we initialize the Terraform context by executing the `terraform init` command.

Then, we perform a check of the code validity by executing the `terraform validate` command. This command checks the syntax of the configuration and returns a list of errors.

There's more...

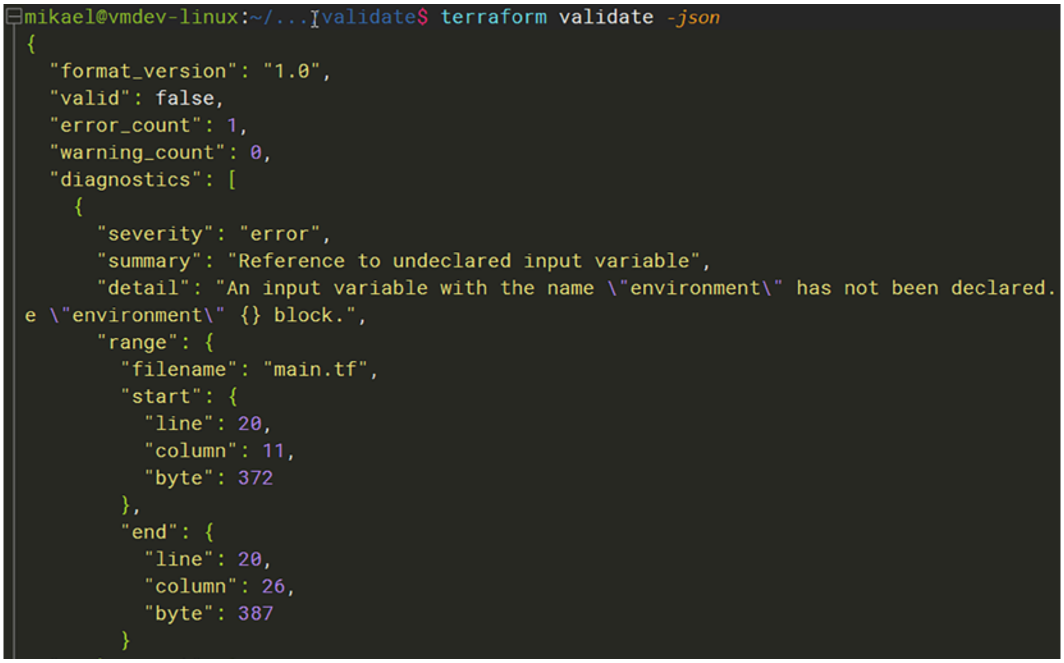
This validation command is useful in local development mode, but also in code integration in a **continuous integration** (CI) pipeline, so as to not execute the `terraform plan` command if the `terraform validate` command returns syntax errors.

The following PowerShell code shows an example of the return code following the execution of this command:

```
terraform validate
$LASTEXITCODE
```

The PowerShell variable `$LASTEXITCODE`, which is native to PowerShell, will return `0` if there is no error, or `1` if there is an error. This is similar to the `$?` variable in many Linux shells.

It is also possible to get the output of the `terraform validate` command in JSON format by adding the `-json` option to this command, as shown in the following screenshot:



```
mikael@vmdev-linux:~/...validate$ terraform validate -json
{
  "format_version": "1.0",
  "valid": false,
  "error_count": 1,
  "warning_count": 0,
  "diagnostics": [
    {
      "severity": "error",
      "summary": "Reference to undeclared input variable",
      "detail": "An input variable with the name \"environment\" has not been declared.
e \"environment\" {} block.",
      "range": {
        "filename": "main.tf",
        "start": {
          "line": 20,
          "column": 11,
          "byte": 372
        },
        "end": {
          "line": 20,
          "column": 26,
          "byte": 387
        }
```

Figure 6.9: The terraform validate output to JSON

The JSON result can then be parsed with third-party tools such as **jq** and used in your workflow.

Be careful, however. This command only performs static validation of the configuration and passing validation; therefore, it does not necessarily imply successful execution (with the `terraform apply` command), as that may depend on other external factors, such as the real state of the infrastructure at the time of the execution of the `terraform apply`. A common problem that is typically not detectable by static analysis is naming conflicts, where the same resource already exists, but Terraform wouldn't know until it reaches the Azure APIs.

If the Terraform configuration contains a `backend` block, then, for this validation of the configuration, we don't need to connect to the state file. We can add the `-backend=false` option to the `terraform init` command.

If you want to perform a more thorough validation that includes variables, you can provide those (e.g. via the `-var` or `-var-file` argument, or via the `tfvars` file) and run `terraform plan`, which will indirectly perform the same validation with variables replaced.

See also

The `terraform validate` command documentation is available here: `https://www.terraform.io/docs/commands/validate.html`.

Destroying infrastructure resources

As we have said many times in this book, IaC allows the rapid provisioning of infrastructure.

Another advantage of IaC is that it allows a quick build and the cleaning up of resources that have been provisioned.

Indeed, we may need to clean up an infrastructure for different reasons. Here are a few examples:

- We destroy an infrastructure with a view to rebuilding it better in accordance with new specifications.
- We provide an infrastructure on demand, which means it is temporary for a specific need (such as to test a new feature or a new branch of the application). And this infrastructure must be capable of being built and destroyed quickly and automatically.
- We want to remove unused infrastructure and, at the same time, no longer pay for it.

In this recipe, we will discuss how to destroy an infrastructure that has been provisioned with Terraform.

Getting ready

To get started, we will provide an infrastructure in Azure that uses Azure App Service.

For this, we use the Terraform configuration that can be found here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/sampleApp`.

To provision it, we execute the basic Terraform workflow with the following commands:

```
terraform init
terraform plan -out="app.tfplan"
terraform apply "app.tfplan"
```

At the end of its execution, we have an Azure resource group, an App Service plan, an App Service instance, and an Application Insights resource in Azure.

The goal of this recipe is to completely destroy this infrastructure using Terraform commands.

How to do it...

To clean your Terraform infrastructure by deleting resources, perform the following steps:

1. To start, initialize the Terraform context by running the `init` command:

   ```
   terraform init
   ```

2. Then, to destroy the resources, we execute the following command:

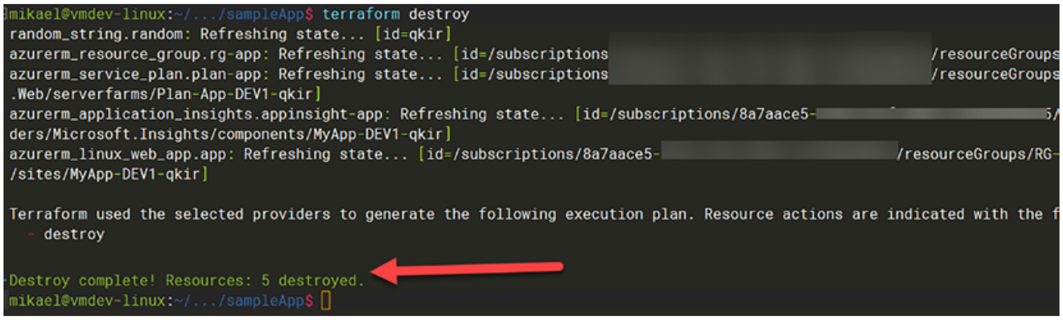
   ```
   terraform destroy
   ```

At the beginning of its execution, this command displays all resources that will be destroyed and asks for confirmation to delete the resources. Validation is then confirmed by typing the word yes.

How it works...

In *Step 1*, we run the `terraform init` command to initialize the Terraform context.

In *Step 2*, we destroy all the provisioned resources by executing the `terraform destroy` command. The following screenshot shows the extracted output of this command:



```
mikael@vmdev-linux:~/.../sampleApp$ terraform destroy
random_string.random: Refreshing state... [id=qkir]
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions[redacted]/resourceGroups
azurerm_service_plan.plan-app: Refreshing state... [id=/subscriptions[redacted]/resourceGroups
.Web/serverfarms/Plan-App-DEV1-qkir]
azurerm_application_insights.appinsight-app: Refreshing state... [id=/subscriptions/8a7aace5-[redacted]5/
ders/Microsoft.Insights/components/MyApp-DEV1-qkir]
azurerm_linux_web_app.app: Refreshing state... [id=/subscriptions/8a7aace5-[redacted]/resourceGroups/RG-
/sites/MyApp-DEV1-qkir]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the f
  - destroy

Destroy complete! Resources: 5 destroyed.
mikael@vmdev-linux:~/.../sampleApp$
```

Figure 6.10: The terraform destroy command execution

At the end of the command execution, Terraform reports that the resources have been successfully destroyed.

There's more...

In this recipe, we studied how to destroy all the resources that have been described and provisioned with a Terraform configuration.

Since the `terraform destroy` command deletes all the resources tracked in the Terraform state, it is important to break the Terraform configuration by separating it into multiple Terraform state to reduce the room for error when changing the infrastructure.

If you need to destroy a single resource while keeping all the resources tracked in the Terraform state, then you can add the `-target` option to the `terraform destroy` command, which allows you to target the resource to be deleted. The following is an example of this command with the `target` option:

```
terraform destroy -target azurerm_application_insights.appinsight-app
```

In this example, only the Application Insights resource is destroyed. For more details, read the pertinent documentation here: `https://www.terraform.io/docs/commands/plan.html#resource-targeting`.

Note that the targeting mechanism should only be used as a last resort. In an ideal scenario, the configuration stays in sync with the Terraform state (as applied without any extra target flags). The risk of executing a targeted apply or destroy operation is that other contributors may miss the context, and more importantly, it becomes much more difficult to apply further changes after changing the configuration.

In addition, if the `terraform plan` command applied in the Terraform configuration requires the `-var-file` option to specify or override values to the variables, then the same options must also be added to the `terraform destroy` command.

In most cases, all options that apply to the `terraform plan` command also apply to the `terraform destroy` command.

See also

- The documentation pertaining to the `terraform destroy` command is available here: `https://www.terraform.io/docs/commands/destroy.html`
- The documentation pertaining to addressing the resource target is available here: `https://www.terraform.io/docs/internals/resource-addressing.html`

Displaying a list of providers used in a configuration

When we use several Terraform providers inside a Terraform configuration, it's important to have governance over the providers list to upgrade them frequently.

In this recipe, we will learn how to display the list of providers used in our Terraform configuration with their versions.

Let's get started!

Getting ready

To complete this recipe, we will use a sample Terraform configuration that contains some providers.

The following code contains the Terraform configuration:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.4.3"
    }
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "3.29.1"
    }
    http = {
      source  = "hashicorp/http"
      version = "3.2.1"
    }
    null = {
      source  = "hashicorp/null"
      version = "3.2.1"
    }
  }
}
```

This Terraform configuration uses the following providers: `random`, `azurerm`, `http`, and `null`.

The goal in this recipe is to list used providers.

The complete source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/providers`.

How to do it...

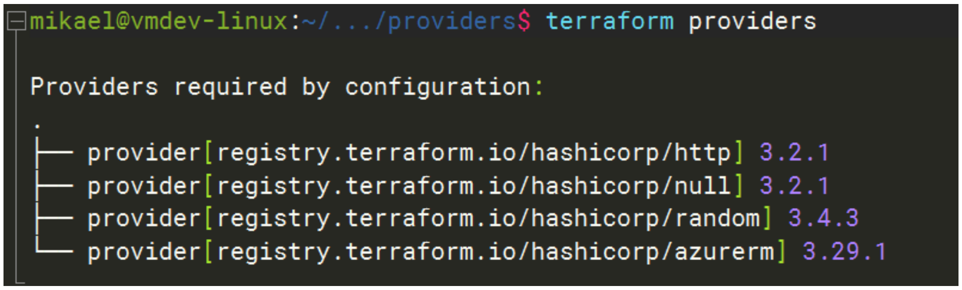
Run the following command at the root of the Terraform configuration to list used providers:

```
terraform providers
```

How it works...

The command `terraform providers` displays the Terraform provider list used in the configuration.

With our sample Terraform configuration, the output of this command is shown in the following image:



```
mikael@vmdev-linux:~/.../providers$ terraform providers

Providers required by configuration:
.
├── provider[registry.terraform.io/hashicorp/http] 3.2.1
├── provider[registry.terraform.io/hashicorp/null] 3.2.1
├── provider[registry.terraform.io/hashicorp/random] 3.4.3
└── provider[registry.terraform.io/hashicorp/azurerm] 3.29.1
```

Figure 6.11: The Terraform providers list in the configuration

We can see all the used Terraform providers with their specified versions in the output above.

There's more...

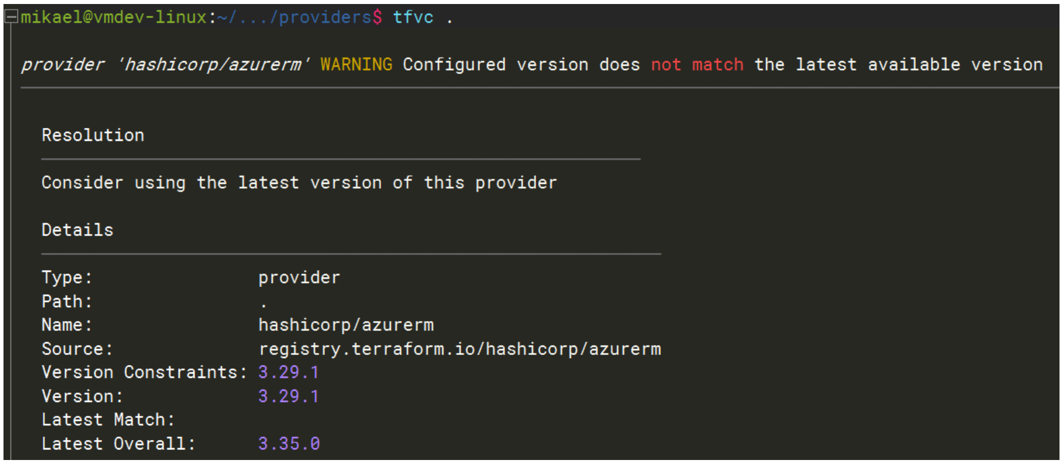
This command is interesting, but it only allows you to list the providers with the versions currently in use (as installed via the last `terraform init` execution).

If you want to go further and know which providers are not up to date and need to be updated, this command does not allow you to do this check.

So, for a better way to check which providers are not up to date, we can use a third-party tool called `tfvc` whose documentation is available here: `https://tfverch.github.io/tfvc/v0.7.12/`. We can install `tfvc` with the following script (on Linux):

```
wget https://github.com/tfverch/tfvc/releases/download/v0.7.12/
tfvc_0.7.12_Linux_x86_64.tar.gz && tar xzvf tfvc_0.7.12_Linux_x86_64.tar.
gz && sudo cp tfvc /usr/local/bin
```

We can then execute `tfvc` at the root of our Terraform configuration with the command `tfvc .`, and we get the following output:



```
mikael@vmdev-linux:~/.../providers$ tfvc .

provider 'hashicorp/azurerm' WARNING Configured version does not match the latest available version

  Resolution

  Consider using the latest version of this provider

  Details

  Type:                provider
  Path:                .
  Name:                hashicorp/azurerm
  Source:              registry.terraform.io/hashicorp/azurerm
  Version Constraints: 3.29.1
  Version:             3.29.1
  Latest Match:
  Latest Overall:      3.35.0
```

Figure 6.12: The Terraform providers list using the tfvc tool

We can see that our `azurerm` Terraform provider isn't the latest version and the latest is `3.35.0`, so we need to update it.

Additionally, we can see the list of providers directly in VS Code using the Terraform extension that we learned about in *Chapter 1*, in the recipe *Writing Terraform configuration in VS Code*. For more information about this feature in the extension read the documentation here: `https://github.com/hashicorp/vscode-terraform#terraform-module-and-provider-explorer`.

See also

- The documentation of the terraform `provider` command is available here: `https://developer.hashicorp.com/terraform/cli/commands/providers`
- The documentation of the `tfvc` tool is available here: `https://tfverch.github.io/tfvc/v0.7.12/`

Generating one Terraform lock file with Windows and Linux compatibility

We previously learned in *Chapter 1*, *Setting Up the Terraform Environment*, in the recipe *Upgrading Terraform providers,* that the Terraform dependency file (`.terraform.lock.hcl`) contains information used by Terraform providers in the Terraform configuration.

Among these providers' information, there are the name, the version, and also the hashes of the packages for integrity checks.

What is important to know is that the package hashes are different, depending on the **operating system** (**OS**) that runs Terraform, since both the Terraform CLI and provider builds (in both cases written in Go) are built for each combination of OS and architecture separately.

And so, the problem you may encounter is that developers work and test their Terraform configuration on a Windows or macOS machine, but the CI pipeline that deploys that same Terraform configuration on other environments is run on a Linux machine.

The Terraform providers that will be downloaded using the command will be based on different OSes and, therefore, the file `.terraform.lock.hcl` will have different hashes. This produces different files than what was sourced in Git.

The goal of this recipe is to show how to generate a `.terraform.lock.hcl` file that contains hashes of used providers for different OSes.

This can also help you ensure that the providers in use are, in fact, available for the given platforms where you intend to use them. For example, not all providers may be available for Windows, or the ARM architecture.

Let's get started!

Getting ready

To complete this recipe, we will use the following Terraform provider configuration:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "3.4.3"
    }
  }
}
```

This Terraform provider configuration uses the `random` provider.

Now we will generate `.terraform.lock.hcl` for both Windows and Linux compatibility.

How to do it...

Perform the following steps to generate `terraform.lock.hcl`:

1. Run the following command on any OS:

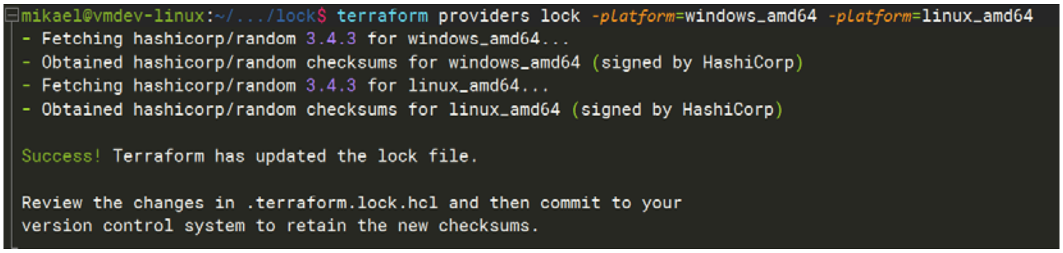
```
terraform providers lock -platform=windows_amd64 -platform=linux_
amd64
```

2. Then, run the basic Terraform workflow to apply our Terraform configuration on a local machine.
3. Finally, we can source `.terraform.lock.hcl` in Git source control.

How it works...

In *Step 1*, we run the `terraform provider lock` command and specify the target OS that will run this Terraform configuration; here, we specify `windows_amd64` and `linux_amd64`.

The following image shows the output of this command:



```
mikael@vmdev-linux:~/.../lock$ terraform providers lock -platform=windows_amd64 -platform=linux_amd64
- Fetching hashicorp/random 3.4.3 for windows_amd64...
- Obtained hashicorp/random checksums for windows_amd64 (signed by HashiCorp)
- Fetching hashicorp/random 3.4.3 for linux_amd64...
- Obtained hashicorp/random checksums for linux_amd64 (signed by HashiCorp)

Success! Terraform has updated the lock file.

Review the changes in .terraform.lock.hcl and then commit to your
version control system to retain the new checksums.
```

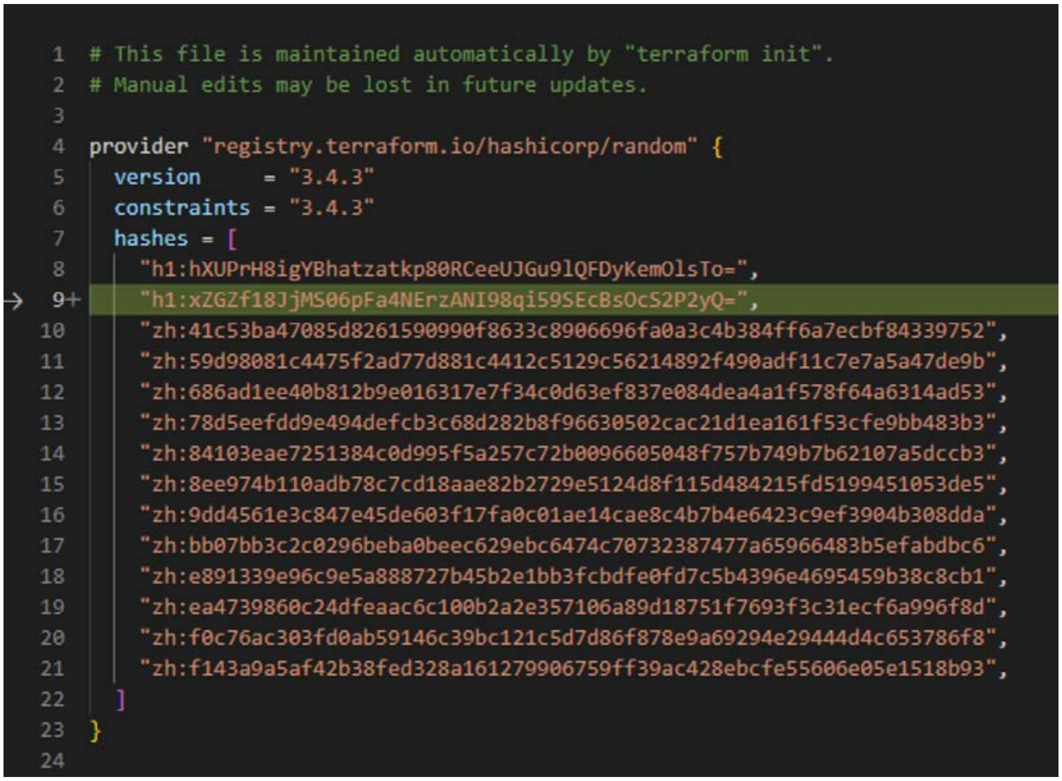
Figure 6.13: The Terraform providers lock command execution

The generated `.terraform.lock.hcl` contains provider hashes for both specified operating systems.

Then, in *Step 2*, we can run this same Terraform configuration locally for example on the Windows OS while ensuring reproducibility through the exact same version of the provider.

Finally, we store this file in Git source control to be executed in the CI pipeline. This file can be run on a Linux OS, and during the `terraform init` execution no changes will be applied to this file.

The following image shows changes applied on `.terraform.lock.hcl`, generated on Windows without any OS specification and then updated with both Windows and Linux OSes.



```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/random" {
  version     = "3.4.3"
  constraints = "3.4.3"
  hashes = [
    "h1:hXUPrH8igYBhatzatkp80RCeeUJGu91QFDyKemOlsTo=",
    "h1:xZGZf18JjMS06pFa4NErzANI98qi59SEcBsOcS2P2yQ=",
    "zh:41c53ba47085d8261590990f8633c8906696fa0a3c4b384ff6a7ecbf84339752",
    "zh:59d98081c4475f2ad77d881c4412c5129c56214892f490adf11c7e7a5a47de9b",
    "zh:686ad1ee40b812b9e016317e7f34c0d63ef837e084dea4a1f578f64a6314ad53",
    "zh:78d5eefdd9e494defcb3c68d282b8f96630502cac21d1ea161f53cfe9bb483b3",
    "zh:84103eae7251384c0d995f5a257c72b0096605048f757b749b7b62107a5dccb3",
    "zh:8ee974b110adb78c7cd18aae82b2729e5124d8f115d484215fd5199451053de5",
    "zh:9dd4561e3c847e45de603f17fa0c01ae14cae8c4b7b4e6423c9ef3904b308dda",
    "zh:bb07bb3c2c0296beba0beec629ebc6474c70732387477a65966483b5efabdbc6",
    "zh:e891339e96c9e5a888727b45b2e1bb3fcbdfe0fd7c5b4396e4695459b38c8cb1",
    "zh:ea4739860c24dfeaac6c100b2a2e357106a89d18751f7693f3c31ecf6a996f8d",
    "zh:f0c76ac303fd0ab59146c39bc121c5d7d86f878e9a69294e29444d4c653786f8",
    "zh:f143a9a5af42b38fed328a161279906759ff39ac428ebcfe55606e05e1518b93",
  ]
}
```

Figure 6.14: The Terraform dependency file updated to a different OS

We can see the added line in the `hashes` property for the Linux provider package.

See also

- The documentation of the Terraform dependency file is available here: `https://developer.hashicorp.com/terraform/language/files/dependency-lock`
- The documentation of the `terraform provider lock` command is available here: `https://developer.hashicorp.com/terraform/cli/commands/providers/lock`

Copying a Terraform module configuration

In *Chapter 3*, *Scaling Your Infrastructure with Terraform*, in the recipe *Provisioning infrastructure in multiple environments*, we learned about different types of folder structures to organize your Terraform configuration.

In this recipe, we will walk through another method to share one Terraform configuration in a shared folder, and we will use the Terraform CLI to retrieve this shared configuration inside our current Terraform configuration.

Let's get started!

Getting ready

To complete this recipe, we will use a sample Terraform configuration that provisions an Azure resource group.

The goal of this recipe is to share this Terraform configuration to a separate and centralized folder and access the configuration from inside that folder by using the init command .

For the shared Terraform configuration, we will reuse the Terraform configuration that we wrote in the previous chapter, the source code of which is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP05/import`.

How to do it…

To share the Terraform configuration, perform the following steps:

1. In the root of the CHAP06 folder, Create a folder called `fromsource` and inside it run the command `init` with the following argument:

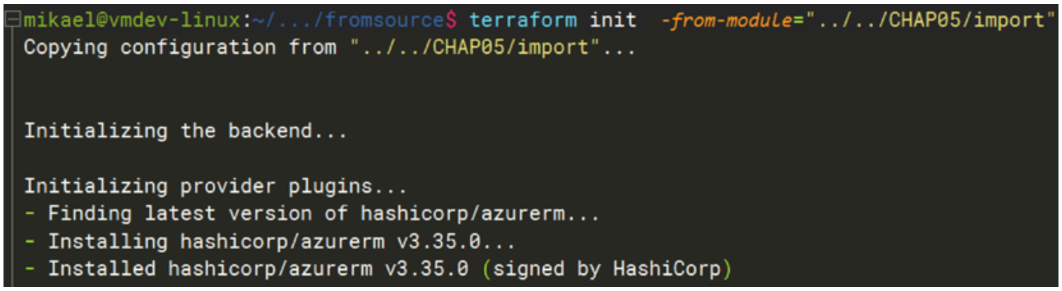
   ```
   terraform init -from-module="../../CHAP05/import"
   ```

2. Run the Terraform workflow with the `init`, `plan`, and `apply` commands.

How it works…

In *Step 1*, in the new folder, we run the command `terraform init -from-module="../../CHAP05/import"` that copies the entire content of the Terraform folder "`import`" to inside the current directory that is empty.

The following image shows the execution of the `init -from-module` command:



```
mikael@vmdev-linux:~/.../fromsource$ terraform init  -from-module="../../CHAP05/import"
Copying configuration from "../../CHAP05/import"...


Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/azurerm...
- Installing hashicorp/azurerm v3.35.0...
- Installed hashicorp/azurerm v3.35.0 (signed by HashiCorp)
```

Figure 6.15: Import shared configuration

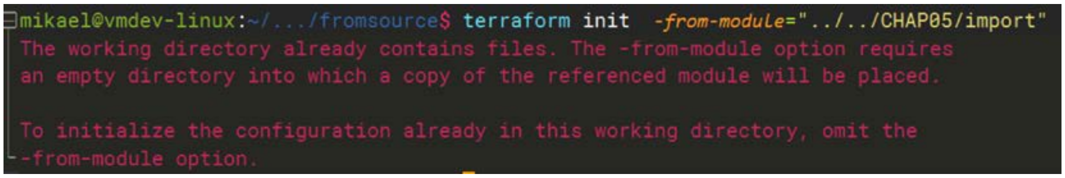
We can see that the content of the import folder is copied into the current directory.

Then, we can run the Terraform workflow inside this `fromsource` directory.

There's more...

The advantage of this method is that we can do other operations after running the `init -from-module` command, for example, copying other files from an external source like `tfvars` files.

Another consideration, and one that we have to be careful of, is that we cannot run this command if the target directory isn't empty. If we run this command twice, we get the following error:



```
mikael@vmdev-linux:~/.../fromsource$ terraform init  -from-module="../../CHAP05/import"
The working directory already contains files. The -from-module option requires
an empty directory into which a copy of the referenced module will be placed.

To initialize the configuration already in this working directory, omit the
-from-module option.
```

Figure 6.16: The import shared configuration error if the folder isn't empty

This error message indicates that the folder isn't empty.

We will learn the best method to share Terraform configurations using modules in *Chapter 7, Sharing Terraform Configurations with Modules*.

See also

- The documentation of the `-from-module` option is available here: `https://developer.hashicorp.com/terraform/cli/commands/init#copy-a-source-module`.

Using workspaces to manage environments

In Terraform, there is the concept of **workspaces**, which enable the same Terraform configuration to be used in order to build multiple environments.

Each of these configurations will be written to a different Terraform state and will, thus, be isolated from the other configurations. Workspaces can be used to create several environments for our infrastructure.

Be careful: in this recipe, we will study the workspaces of Terraform states, also called Terraform CLI workspaces, which are different from the Terraform Cloud workspaces used in HCP, which we will study in *Chapter 14*, *Using Terraform Cloud to Improve Team Collaboration*, in the recipe *Using workspaces in Terraform Cloud*.

In this recipe, we will study the use of Terraform workspaces in Terraform configurations, with the execution of Terraform CLI commands.

Getting ready

The purpose of this recipe is for an application to create an Azure resource group for each of its environments (`dev` and `prod`).

Regarding the Terraform configuration, there are no prerequisites, as we will see in the steps of the recipe.

The Terraform configuration for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/workspaces`.

How to do it…

To manage a Terraform workspace, perform the following steps:

1. In a new `main.tf` file, we write the following Terraform configuration:

```
resource "azurerm_resource_group" "rg-app" {
  name     = "RG-APP-${terraform.workspace}"
  location = "westeurope"
}
```

2. In a command terminal, we navigate into the folder that contains this Terraform configuration and execute the following command:

```
terraform init
terraform workspace new dev
```

3. To provision the dev environment, we run the basic commands of the Terraform workflow, which are as follows:

```
terraform plan -out="outdev.tfplan"
terraform apply "outdev.tfplan"
```

4. Then, we execute the `workspace new` command with the name of the production workspace to be created:

```
terraform workspace new prod
```

5. To finish and provision the `prod` environment, we execute the basic commands of the Terraform workflow production, which are as follows:

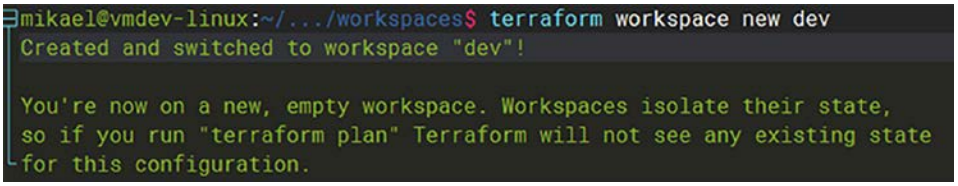
```
terraform plan -out="outprod.tfplan"
terraform apply "outprod.tfplan"
```

How it works...

In *Step 1*, in the Terraform configuration we wrote, we provide a resource group in Azure that will have a name composed of an `RG-APP` prefix and a dynamic suffix, `terraform.workspace`, which will be the name of the workspace we are going to create.

In *Step 2*, we create the workspace that corresponds to the `dev` environment, and for this, we use the `terraform workspace new` command followed by the workspace name (in this case, `dev`).

Once created, Terraform automatically switches to this workspace, as you can see in the following screenshot:



```
mikael@vmdev-linux:~/.../workspaces$ terraform workspace new dev
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

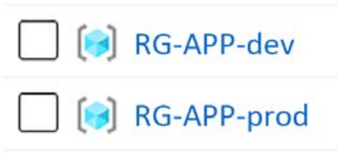
Figure 6.17: The Terraform created workspace

After we've created the workspace, we just execute the basic commands of the Terraform workflow, which we do in *Step 3*.

> Note that here we have added the `-out` option to the `terraform plan` command to save the result of the plan in the `outdev.tfplan` file. Then, to apply the changes, we specifically add this file as an argument to the `terraform apply` command.
>
> This option is important in Terraform automation mode, to ensure that what will be applied by Terraform is equal to the `plan` and nobody can change the Terraform configuration between the execution of the `plan` and `apply` commands.

Then, to provision the `prod` environment, we repeat *Steps 2* and *3*, but this time, we create a workspace called `prod`.

At the end of the execution of all these steps, we can see in the Azure portal that we have our two resource groups, which contain in their suffixes the names of their workspaces, as you can see in the following screenshot:

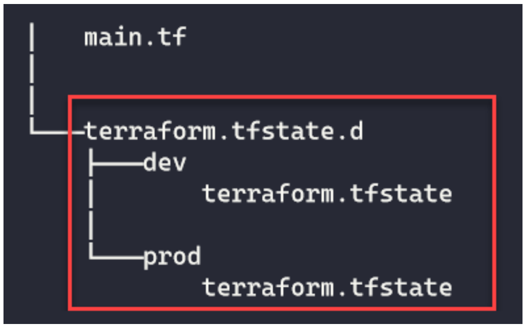


RG-APP-dev

RG-APP-prod

Figure 6.18: Azure resource groups created with the Terraform workspace suffix

In addition, we also notice two Terraform states, one for each workspace, which were created automatically, as shown in the following screenshot:



```
│   main.tf
│
│
└──terraform.tfstate.d
    ├──dev
    │       terraform.tfstate
    │
    └──prod
            terraform.tfstate
```

Figure 6.19: One Terraform state for each workspace

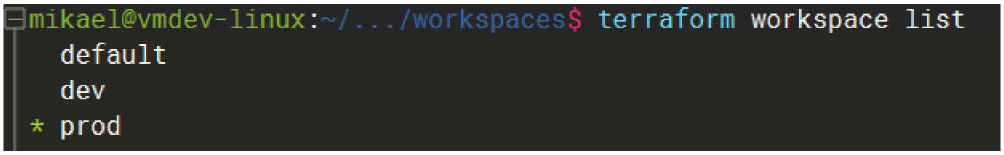
In this screenshot, we can see two `terraform.tfstate` files, one in the `dev` directory and another in the `prod` directory.

There's more...

It is possible to see the list of workspaces in the backend by executing the following command:

```
terraform workspace list
```

The following screenshot shows the execution of this command in the case of our recipe:



```
mikael@vmdev-linux:~/.../workspaces$ terraform workspace list
  default
  dev
* prod
```

Figure 6.20: A list of Terraform workspaces

In any Terraform configuration execution, there is a default workspace that is called `default`, which we can also see in the list above. We can clearly see our `dev` and `prod` workspace, and that the current workspace is `prod` (marked with an * in front of its name).

If you want to switch to another workspace, execute the `terraform workspace select` command, followed by the name of the workspace to be selected – for example:

```
terraform workspace select dev
```

Then, before using Terraform workspace, check if your Terraform state backend is compatible with multiple workspaces by referring to this page: `https://developer.hashicorp.com/terraform/language/state/workspaces#backends-supporting-multiple-workspaces`.

Finally, you can also delete a workspace by executing the `terraform workspace delete` command, followed by the name of the workspace to be deleted – for example:

```
terraform workspace delete dev
```

Please note that, when deleting a workspace, the associated resources are not deleted. That's why, in order to delete a workspace, you must first delete the resources provided by that workspace using the `terraform destroy` command. Otherwise, if this operation is not carried out, it will no longer be possible to manage these resources with Terraform because the Terraform state of this workspace will have been deleted.

For the reasons described above, as a precaution, it is not possible to delete a workspace whose Terraform state is not empty by default. However, we can force the destruction of this workspace by adding the `-force` option to the `terraform workspace delete -force` command, as documented here: `https://www.terraform.io/docs/commands/workspace/delete.html`.

See also

- The general documentation for workspaces is available here: `https://www.terraform.io/docs/state/workspaces.html`
- The CLI documentation for the `terraform workspace` command is available here: `https://www.terraform.io/docs/commands/workspace/index.html`
- Read this blog post for a more complete understanding of workspaces: `https://www.colinsalmcorner.com/terraform-all-the-things-with-vsts/`

Exporting the output in JSON

In the *Looping over object collections* and *Using outputs to expose Terraform provisioned data* recipes of *Chapter 2*, *Writing Terraform Configurations*, we discussed the use of Terraform's outputs, which allow you to have output values for the execution of the Terraform configuration.

Indeed, we have seen how to declare an output in the Terraform configuration, and we learned that these outputs and their values were displayed at the end of the execution of the `terraform apply` command.

The advantage of these outputs is that they can be retrieved by another program and, thus, be used for another operation – for example, in a CI/CD pipeline.

In this recipe, we will see how the values of the outputs can be retrieved in JSON format so that they can be used in an external program.

Getting ready

For this recipe, we will use only the Terraform configuration that we already studied in *Chapter 3*, *Scaling Your Infrastructure with Terraform*, and whose sources can be found here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP03/list_map`.

In this code, we add another output that returns the list of Azure app services URLs as shown:

```
output "app_service_urls" {
  value = {for x in azurerm_app_service.app :  x.name => x.default_site_
hostname  }
}
```

To explore the values of the output, we need to use a tool that allows us to work with JSON. For this, you can use any framework and library according to your scripting languages. In this recipe, we will use **jq**, which is a free tool that allows you to easily manipulate JSON on the command line. The documentation on jq installation is available here: `https://stedolan.github.io/jq/`.

The purpose of this recipe is to provision two Azure App Service instances using a Terraform configuration and then, with a script, perform a response check of the URL of the first App Service instance.

How to do it...

Perform the following steps to export Terraform output for use in other programs:

1. Execute the Terraform workflow with the following commands:

```
terraform init
terraform plan -out="app.tfplan
"terraform apply "app.tfplan"
```

2. Then, run the `terraform output` command:

```
terraform output
```

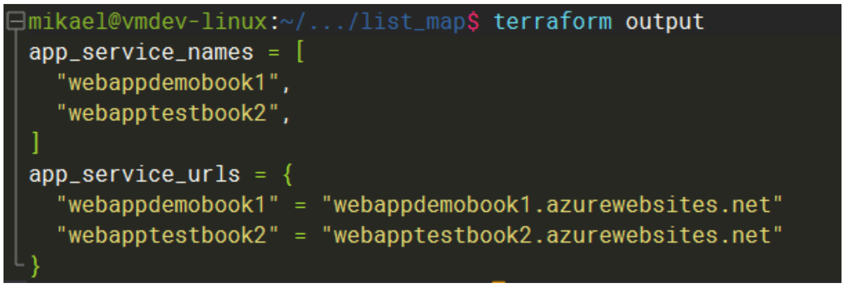
3. Finally, to retrieve the URL of the created App Service instance, we execute the following command in the command terminal:

```
urlwebapp1=$(terraform output -json | jq -r .app_service_urls.value.
webappdemobook1)
```

How it works...

In *Step 1*, we execute the basic commands of the Terraform workflow. After executing the `terraform apply` command, the command displays the output.

Then, in *Step 2*, we visualize the output of this Terraform configuration more clearly by executing the `terraform output` command, as shown in the following screenshot:



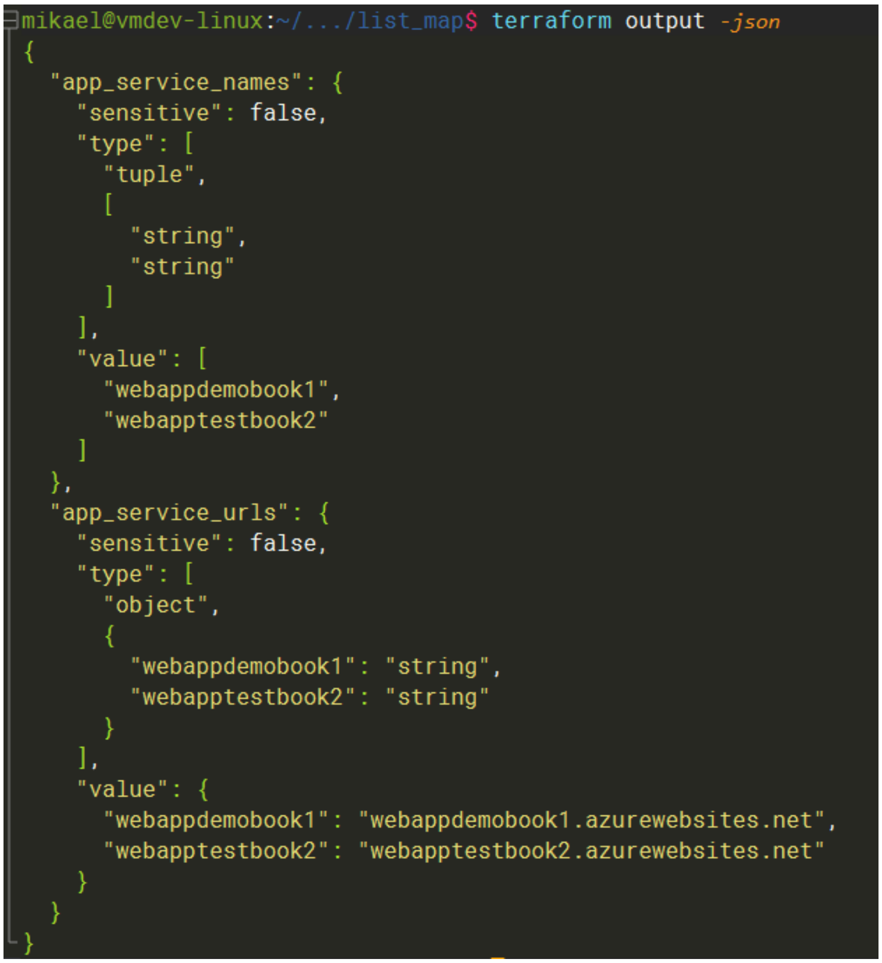
```
mikael@vmdev-linux:~/.../list_map$ terraform output
app_service_names = [
  "webappdemobook1",
  "webapptestbook2",
]
app_service_urls = {
  "webappdemobook1" = "webappdemobook1.azurewebsites.net"
  "webapptestbook2" = "webapptestbook2.azurewebsites.net"
}
```

Figure 6.21: The result of the terraform output command

the preceding screenshot, we can see that this command returns the two outputs declared in the code, which are as follows:

- `app_service_names`: This returns a list of App Service names.
- `app_service_urls`: This returns a list of the URLs of provisioned App Service instances.

Finally, in *Step 3*, we run a script that checks the URL of the `webappdemobook1` App Service instance. In the first line of this script, we execute the `terraform output -json` command, which enables the result of the output to be returned in JSON format, as you can see in the following screenshot:



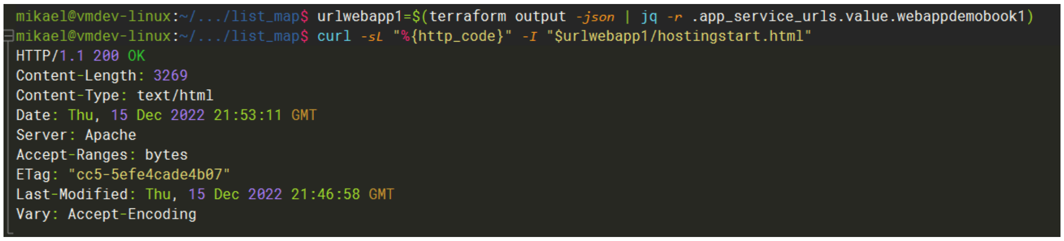
```
mikael@vmdev-linux:~/.../list_map$ terraform output -json
{
  "app_service_names": {
    "sensitive": false,
    "type": [
      "tuple",
      [
        "string",
        "string"
      ]
    ],
    "value": [
      "webappdemobook1",
      "webapptestbook2"
    ]
  },
  "app_service_urls": {
    "sensitive": false,
    "type": [
      "object",
      {
        "webappdemobook1": "string",
        "webapptestbook2": "string"
      }
    ],
    "value": {
      "webappdemobook1": "webappdemobook1.azurewebsites.net",
      "webapptestbook2": "webapptestbook2.azurewebsites.net"
    }
  }
}
```

Figure 6.22: terraform output in JSON format

Then, with this result in JSON, we use the **jq** tool on it by retrieving the URL of the `webappdemobook1` App Service instance. The returned URL is put in a variable called `urlwebapp1`.

Then, in the second line of this script, we use cURL on this URL by passing options to return only the HTTP header of the URL.

The result of executing this script is shown in the following screenshot:



```
mikael@vmdev-linux:~/.../list_map$ urlwebapp1=$(terraform output -json | jq -r .app_service_urls.value.webappdemobook1)
mikael@vmdev-linux:~/.../list_map$ curl -sL "%{http_code}" -I "$urlwebapp1/hostingstart.html"
HTTP/1.1 200 OK
Content-Length: 3269
Content-Type: text/html
Date: Thu, 15 Dec 2022 21:53:11 GMT
Server: Apache
Accept-Ranges: bytes
ETag: "cc5-5efe4cade4b07"
Last-Modified: Thu, 15 Dec 2022 21:46:58 GMT
Vary: Accept-Encoding
```

Figure 6.23: Terraform output after using the curl command

You can see that the result of the check is `OK`, with a status code of `200`.

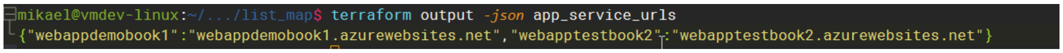
There's more...

In this recipe, we learned how to retrieve all the `output` of a Terraform configuration. It is also possible to retrieve the value of a particular output by executing the `terraform output <output name>` command.

In our case, we could have executed the `app_service_urls` command to display the value of the output in JSON format:

```
terraform output -json app_service_urls
```

The following screenshot shows the execution of this command:



```
mikael@vmdev-linux:~/.../list_map$ terraform output -json app_service_urls
{"webappdemobook1":"webappdemobook1.azurewebsites.net","webapptestbook2":"webapptestbook2.azurewebsites.net"}
```

Figure 6.24: terraform output execution with the target resource

Then, we would run the following command to check the URL:

```
urlwebapp1=$(terraform output -json app_service_urls | jq -r
.webappdemobook1) &&
curl -sL  -I "$urlwebapp1/hostingstart.html"
```

We can see in this script that the command used is `terraform output -json app_service_urls`, which is more simplistic than `$(terraform output -json | jq -r .app_service_urls.value.webappdemobook1)`.

See also

- The `terraform  output` command documentation is available here: `https://www.terraform.io/docs/commands/output.html`
- The `jq` website documentation can be found here: `https://stedolan.github.io/jq/`

Tainting resources

Earlier, in the *Destroying infrastructure resources* recipe of this chapter, we learned how to destroy resources that have been provisioned with Terraform.

However, in certain situations, you may need to destroy a particular resource just to rebuild it immediately. Examples of such situations may include modifications that have been made manually to that resource.

To destroy and rebuild a resource, you could perform the `terraform destroy -target <resource>` command, followed by the `apply` command. However, the problem is that between the `destroy` and `apply` commands, there may be other undesirable changes applied in the Terraform configuration.

So, in this recipe, we will see how to perform this operation using the Terraform concept of tainting.

Getting ready

In order to apply this recipe, we first provision an infrastructure composed of a resource group, an App Service plan, a Linux App Service instance, and an Application Insights resource. The Terraform configuration used for this provisioning can be found here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/sampleApp`.

The goal of this recipe is to destroy and then rebuild the App Service instance in a single operation, using the `taint` command of Terraform.

How to do it…

To apply the `taint` operation, perform the following steps:

1. Run the `terraform  init`, `plan` and `apply` commands to create the resources
2. Then, execute the `terraform  taint` command to flag the resource as *tainted*:

```
terraform taint azurerm_linux_web_app.app
```

3. Finally, to recreate the App Service instance, execute the following command:

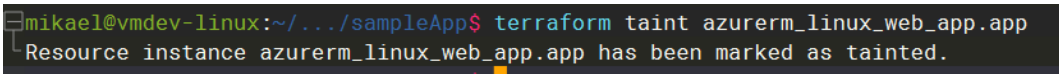
```
terraform apply
```

How it works...

In *Step 1*, we execute the `terraform init` command to initialize the context. Then, in *Step 2*, we execute the `terraform taint` command to flag the `azurerm_app_service.app` resource as tainted – that is, to be destroyed and rebuilt.

This command does not affect the resource itself, but only marks it as tainted in the Terraform state.

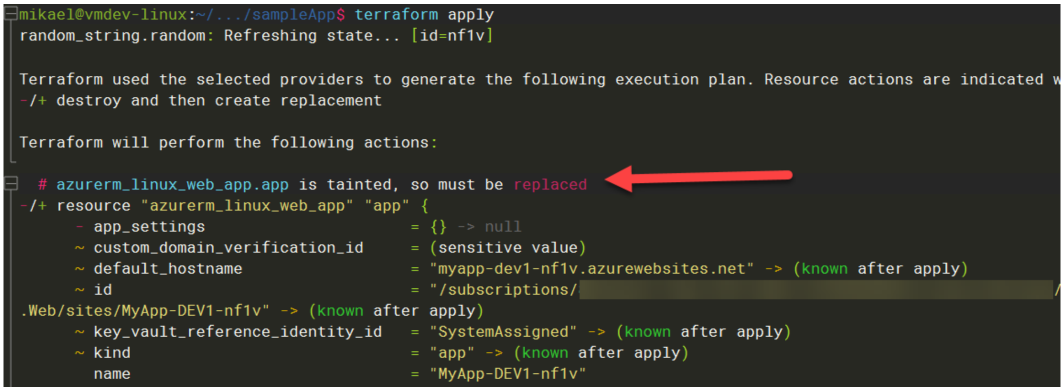
The following screenshot shows the result of the `taint` command:



```
mikael@vmdev-linux:~/.../sampleApp$ terraform taint azurerm_linux_web_app.app
Resource instance azurerm_linux_web_app.app has been marked as tainted.
```

Figure 6.25: The terraform taint command

Finally, in *Step 3*, we execute the `terraform apply` command, and when it is executed, we can see that Terraform will delete and then recreate the Azure Linux App Service instance, as shown in the following screenshot:



```
mikael@vmdev-linux:~/.../sampleApp$ terraform apply
random_string.random: Refreshing state... [id=nf1v]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated w
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # azurerm_linux_web_app.app is tainted, so must be replaced
-/+ resource "azurerm_linux_web_app" "app" {
      - app_settings                       = {} -> null
      ~ custom_domain_verification_id      = (sensitive value)
      ~ default_hostname                   = "myapp-dev1-nf1v.azurewebsites.net" -> (known after apply)
      ~ id                                 = "/subscriptions/[redacted]/
.Web/sites/MyApp-DEV1-nf1v" -> (known after apply)
      ~ key_vault_reference_identity_id    = "SystemAssigned" -> (known after apply)
      ~ kind                               = "app" -> (known after apply)
        name                               = "MyApp-DEV1-nf1v"
```

Figure 6.26: terraform apply after taint

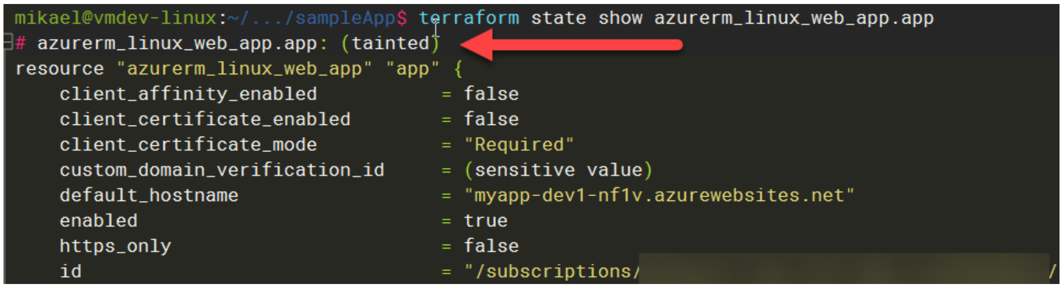
We can see in the preceding screenshot that Terraform destroys the Linux App Service resource and then recreates it.

There's more...

To check that the resource is tainted, we can display the status of this resource in the terminal, flagged in the Terraform state by executing the `terraform state show` command. For a refresher, refer to what we learned in *Chapter 5*, *Managing Terraform State*, in the recipe *Managing resources in the Terraform State*, which displays the contents of the Terraform state in the command terminal (documented here: `https://www.terraform.io/docs/commands/state/show.html`) in response to the following command:

```
terraform state show azurerm_linux_web_app.app
```

The following screenshot shows the result of this command:



```
mikael@vmdev-linux:~/.../sampleApp$ terraform state show azurerm_linux_web_app.app
# azurerm_linux_web_app.app: (tainted)
resource "azurerm_linux_web_app" "app" {
    client_affinity_enabled            = false
    client_certificate_enabled         = false
    client_certificate_mode            = "Required"
    custom_domain_verification_id      = (sensitive value)
    default_hostname                   = "myapp-dev1-nf1v.azurewebsites.net"
    enabled                            = true
    https_only                         = false
    id                                 = "/subscriptions/[redacted]/
```

Figure 6.27: Displaying the status of the resource in the Terraform state

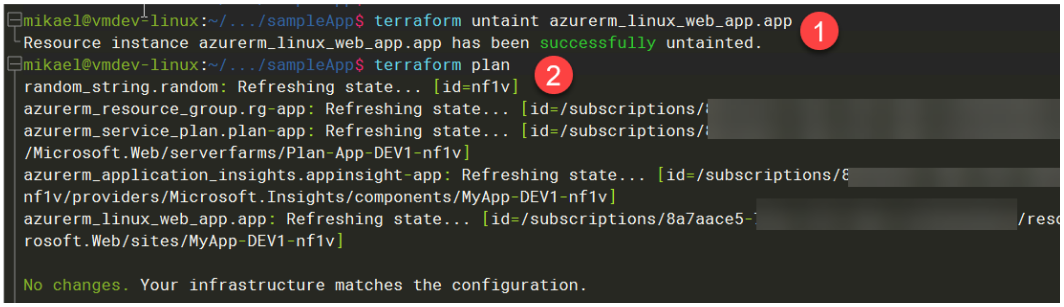
We can see that the App Service resource has the flag `tainted`.

We have used the `terraform state` command to display the contents of the Terraform state, since it is strongly discouraged to read and modify the Terraform state manually, as documented here: `https://www.terraform.io/docs/state/index.html#inspection-and-modification`.

Moreover, in order to cancel the `taint` flag applied with the `terraform taint` command, we can execute the inverse command, which is `terraform untaint`. This command can be executed like this:

```
terraform untaint azurerm_linux_web_app.app
```

Then, if we execute the `terraform plan` command, we can see that there is no change, as shown in the following screenshot:



```
mikael@vmdev-linux:~/.../sampleApp$ terraform untaint azurerm_linux_web_app.app
Resource instance azurerm_linux_web_app.app has been successfully untainted.
mikael@vmdev-linux:~/.../sampleApp$ terraform plan
random_string.random: Refreshing state... [id=nf1v]
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/[redacted]
azurerm_service_plan.plan-app: Refreshing state... [id=/subscriptions/[redacted]
/Microsoft.Web/serverfarms/Plan-App-DEV1-nf1v]
azurerm_application_insights.appinsight-app: Refreshing state... [id=/subscriptions/[redacted]
nf1v/providers/Microsoft.Insights/components/MyApp-DEV1-nf1v]
azurerm_linux_web_app.app: Refreshing state... [id=/subscriptions/8a7aace5-[redacted]/reso
rosoft.Web/sites/MyApp-DEV1-nf1v]

No changes. Your infrastructure matches the configuration.
```

Figure 6.28: The terraform untaint command and plan

We can see in this screenshot that the `untaint` command has canceled the effect of the `taint` command, and during the execution of the `plan` command, no changes will be applied to the infrastructure.

See also

- The `terraform taint` command documentation is available here: `https://www.terraform.io/docs/commands/taint.html`
- The `terraform untaint` command documentation is available here: `https://www.terraform.io/docs/commands/untaint.html`
- The `terraform state` command documentation is available here: `https://www.terraform.io/docs/commands/state/index.html`
- An article that explains the `taint` and `untaint` commands really well can be found here: `https://www.devopsschool.com/blog/terraform-taint-and-untaint-explained-with-example-programs-and-tutorials/`

Generating the dependency graph

One of the interesting features of Terraform is the ability to generate a dependency graph of the resource dependencies declared in the Terraform configuration.

The visualization of the dependency graph is important to understand the dependencies of Terraform resources and the Terraform operations order.

In this recipe, we will see how to generate and visualize this dependency graph.

Getting ready

For this recipe, we need to use a third-party drawing generation tool called **Graphviz**, which is available for download at `https://graphviz.gitlab.io/download/`. You will need to download and install the package corresponding to your OS.

As an example, we will take the Terraform configuration available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/sampleApp`.

How to do it...

To generate the dependency graph, perform the following steps:

1. Inside the folder that contains the Terraform configuration, run the `terraform init` command.
2. Execute the `terraform graph` command:

   ```
   terraform graph | dot -Tsvg > graph.svg
   ```

   On Windows, an encoding issue can occur on the PowerShell Terminal console; however, this command works fine on the Cmd terminal console.

3. Open File Explorer, navigate inside the folder that contains the Terraform configuration, and open the file called `graph.svg`.

How it works...

In *Step 1*, we will execute the `terraform graph` command. Then, we send the result of this graph command to the `dot` utility that was previously installed with Graphviz. This `dot` utility will generate a `graph.svg` file, which contains the graphical representation of the Terraform configuration.

In *Step 2*, we open the `graph.svg` file, and we can see the dependency graph as shown in the following diagram:

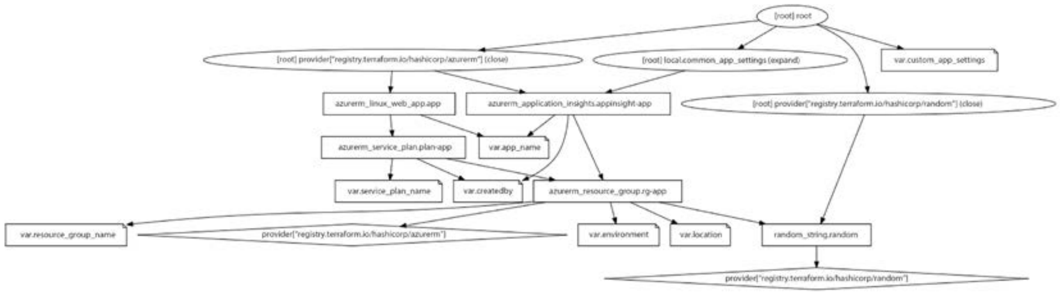

[root] root
[root] provider["registry.terraform.io/hashicorp/azurerm"] (close)
[root] local.common_app_settings (expand)
var.custom_app_settings
azurerm_linux_web_app.app
azurerm_application_insights.appinsight-app
[root] provider["registry.terraform.io/hashicorp/random"] (close)
azurerm_service_plan.plan-app
var.app_name
var.service_plan_name
var.createdby
azurerm_resource_group.rg-app
var.resource_group_name
provider["registry.terraform.io/hashicorp/azurerm"]
var.environment
var.location
random_string.random
provider["registry.terraform.io/hashicorp/random"]

Figure 6.29: The Terraform dependency graph

In the preceding diagram, we can see the dependencies between variables, resources, and the provider.

See also

- The `terraform graph` command documentation is available here: `https://www.terraform.io/docs/commands/graph.html`
- Documentation relating to Graphviz is available here: `https://graphviz.gitlab.io/`

Using different Terraform configuration directories

Until now, in all the recipes we have covered, we executed the Terraform workflow commands in the folder that contains the Terraform configuration to apply.

However, in IaC enterprise scenarios, the Terraform configuration is often separated into multiple folders (see the note below), and to apply the changes we need to navigate to each folder in the correct order.

It is best practice to separate Terraform configurations into multiple folders (also called "modules" or "components"), to isolate the Terraform state, and to have a different life cycle deployment for each component.

In this recipe, we will learn how to run the Terraform CLI and target the Terraform configuration.

Let's get started!

Getting ready

To illustrate this recipe, we will use a Terraform configuration folder structure that deploys a network infrastructure, then deploys the database, and finally, deploys the web infrastructure components.

The following image shows the Terraform folder structure, annotated with the order of the deployment:

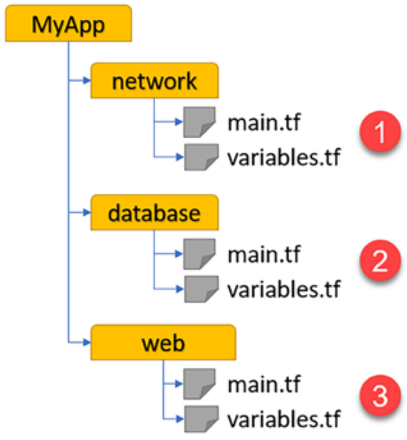


Figure 6.30: The Terraform folder components structure

First of all, to deploy these three components we would run the following script from the `MyApp` folder:

```
cd network
terraform init
terraform plan -out=network.tfplan
terraform apply network.tfplan
cd ..
cd database
terraform init
terraform plan -out=database.tfplan
terraform apply database.tfplan
cd ..
cd web
terraform init
terraform plan -out=web.tfplan
terraform apply web.tfplan
```

As we can see in the previous script we have to navigate in the folders several times for each of the components with `cd ..` and `cd <folder name>`, which could be more complex if we had sub-files at several levels.

This problem of folder navigation is all the more important in an automation context, such as a CI/CD pipeline for Terraform execution.

The goal of this recipe is to learn how to simplify the Terraform execution by switching the Terraform configuration folder with the Terraform CLI.

The sample of this Terraform folder structure with sample configuration is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/MyApp`.

How to do it…

In the `MyApp` folder, run the following script:

```
terraform -chdir=network init
terraform -chdir=network plan -out=network.tfplan
terraform -chdir=network apply network.tfplan

terraform -chdir=database init
terraform -chdir=database plan -out=database.tfplan
terraform -chdir=database apply database.tfplan

terraform -chdir=web init
terraform -chdir=web plan -out=database.tfplan
terraform -chdir=web apply database.tfplan
```

How it works…

In the script running in this recipe, we execute the Terraform workflow from one folder (here, it is the `MyApp` folder), and we add the option `-chdir` to all Terraform commands to target and switch the Terraform configuration.

There's more…

Contrary to the other Terraform options, which are placed after the name of the command to be executed (`terraform <command name> options`), the `-chdir` option is placed before the name of the command (`terraform -chdir <command name>`).

See also

- The documentation of the `-chdir` option is available here: `https://developer.hashicorp.com/terraform/cli/commands#switching-working-directory-with-chdir`

Testing and evaluating a Terraform expression

When we write a Terraform configuration that contains variables and expressions or uses a built-in Terraform function, it's sometimes difficult to predict the result of this Terraform configuration before running the `apply` command.

Fortunately, the Terraform CLI contains a feature that will allow us to evaluate Terraform code before applying changes.

In this recipe, we will see how to test and evaluate a Terraform expression.

Let's get started!

Getting ready

To complete this recipe, we will use the following Terraform configuration that provisions Azure infrastructure:

`https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/console`

Here we use the `azurerm` provider, but this recipe can be applied for any Terraform configuration.

In this Terraform configuration, we use several Terraform variables and expressions such as:

```
locals {
  linux_web_app   = toset([for each in var.web_apps : each.name if each.os
== "Linux"])
  windows_web_app = toset([for each in var.web_apps : each.name if each.os
== "Windows"])
    default_app_settings = {
    "DEFAULT_KEY1" = "DEFAULT_VAL1"
  }
}
```

Or also:

```
app_settings = merge(local.default_app_settings, var.custom_app_settings)
```

The goal of this recipe is to display the result of the expressions:

```
    linux_web_app   = toset([for each in var.web_apps : each.name if each.
  os == "Linux"])
```

and

```
app_settings = merge(local.default_app_settings, var.custom_app_settings)
```

We want to display these results before running the plan and apply commands. That way, we can test that the expression in the Terraform configuration is what we expect.

How to do it...

Perform the following steps to evaluate the Terraform configuration expression:

1. Initialize the Terraform context by running the following command:

   ```
   terraform init -backend=false
   ```

2. Then run the command `console` as follows:

   ```
   terraform console
   ```

3. In the `terraform console` execution, enter the following input to evaluate the expression:

   ```
   local.linux_web_app
   ```

4. Then run the second expression to evaluate with this input:

   ```
   merge(local.default_app_settings, var.custom_app_settings)
   ```

5. Finally, to finish the evaluation, input `exit` to stop the console mode.

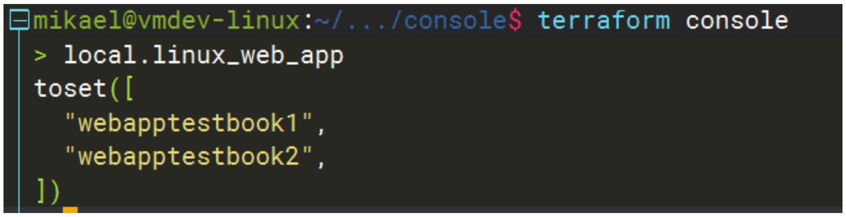
How it works...

In *Step 1*, we run the command `terraform init -backend=false` to initialize the Terraform context without using the backend, because our goal now isn't to apply changes but just to evaluate an expression in the Terraform configuration.

In *Step 2*, we run the command `terraform console`, which opens the console mode.

Then, in the execution of `terraform console`, we can evaluate the desired expressions, displaying the content or values of local expressions.

The following image shows the evaluation of the `local. linux_web_app` local expression:

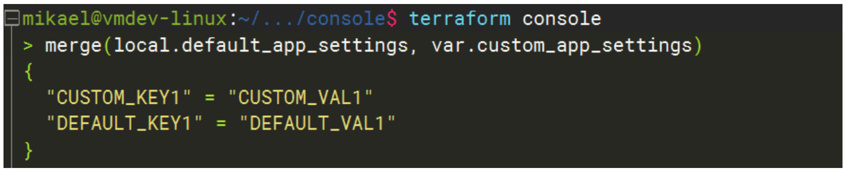


```
mikael@vmdev-linux:~/.../console$ terraform console
> local.linux_web_app
toset([
  "webapptestbook1",
  "webapptestbook2",
])
```

Figure 6.31: Terraform evaluating the locals expression

We can see the content of the list `local.linux_web_app` that will be applied by Terraform.

Then we input the second expression, and the following image shows this evaluation result:



```
mikael@vmdev-linux:~/.../console$ terraform console
> merge(local.default_app_settings, var.custom_app_settings)
{
  "CUSTOM_KEY1" = "CUSTOM_VAL1"
  "DEFAULT_KEY1" = "DEFAULT_VAL1"
}
```

Figure 6.32: Terraform evaluating the built-in function

We can see the expression that contains the built-in function, which in this case is a `merge` function.

We can also evaluate the Terraform variable defined in our Terraform configuration – for example, to evaluate the value of the variable `web_apps`, by inputting the expression `var.web_apps` in the terraform console execution.

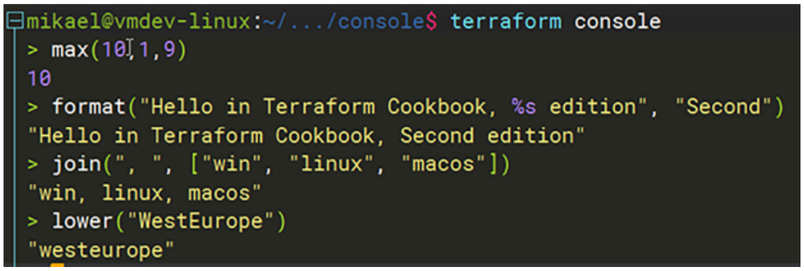
Finally, to close the `console` mode, input the command `exit`.

There's more…

In this recipe, we learned how to use the `terraform console` to evaluate local variables or expressions used in our Terraform configuration.

We can also use this `console` command to evaluate expressions that are not written in our Terraform configuration – for example, to evaluate or test expressions before using them in the configuration, or to learn Terraform built-in functions.

The following screenshot shows some samples of built-in Terraform function evaluation:



```
mikael@vmdev-linux:~/.../console$ terraform console
> max(10,1,9)
10
> format("Hello in Terraform Cookbook, %s edition", "Second")
"Hello in Terraform Cookbook, Second edition"
> join(", ", ["win", "linux", "macos"])
"win, linux, macos"
> lower("WestEurope")
"westeurope"
```

Figure 6.33: Terraform testing the built-in functions

We can see in the image above the evaluation of the `max`, `format`, `join`, and `lower` functions.

See also

- The documentation of the `terraform console` command with more examples is available here: `https://developer.hashicorp.com/terraform/cli/commands/console`
- The learning lab of the `console` command is available here: `https://developer.hashicorp.com/terraform/tutorials/cli/console`

Debugging the Terraform execution

When we execute Terraform commands, the console output is quite simple and clear.

In this recipe, we will study how to activate debug mode in Terraform, which will allow us to display more information about it and trace its execution.

Getting ready

For this recipe, we will use the Terraform configuration available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP06/sampleApp`.

Furthermore, for the purposes of this demonstration, we will run it on a Windows OS, but the operation is exactly the same on other OSes.

How to do it…

To activate the debug on Terraform, perform the following steps:

1. In the terminal console or script, set the `TF_LOG` environment variable by executing the following command:

```
$env:TF_LOG = "TRACE"
```

2. Now, we can execute the Terraform workflow commands with the display logs activated:

```
terraform init
terraform plan -out="out.tfplan"app.tfplan
terraform apply app.tfplan
```

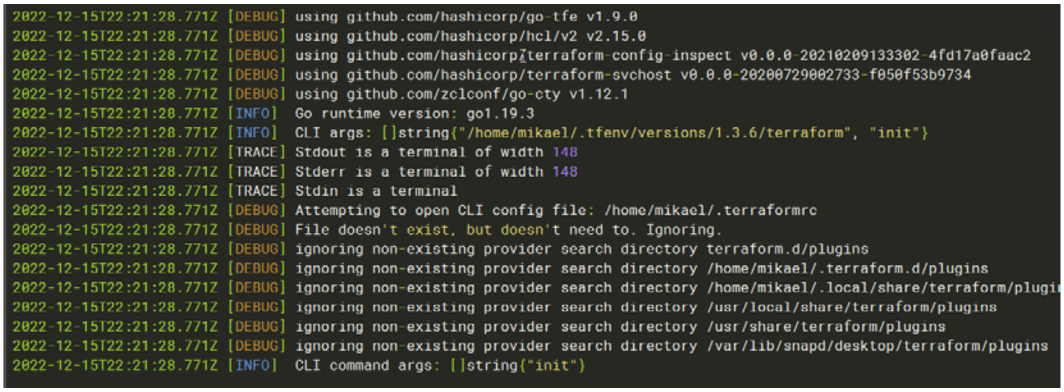
How it works...

In *Step 1*, we create a Terraform environment variable, `TF_LOG`, which enables Terraform's verbose mode to be activated, indicating that we want to see all traces of Terraform's execution displayed.

The possible values we can set this environment variable to are `TRACE`, `DEBUG`, `INFO`, and `WARN`.

> In this recipe, we used the `$env` command to set this environment variable because we are working on Windows. You can, of course, do the same on other OSes with the correct syntax.
>
> For example, in a shell, script execute `export TF_LOG="TRACE"`.

Then, in *Step 2*, we execute the commands of the Terraform workflow, and we can see in the output all traces of this execution, as shown in the following screenshot:



```
2022-12-15T22:21:28.771Z [DEBUG] using github.com/hashicorp/go-tfe v1.9.0
2022-12-15T22:21:28.771Z [DEBUG] using github.com/hashicorp/hcl/v2 v2.15.0
2022-12-15T22:21:28.771Z [DEBUG] using github.com/hashicorp/terraform-config-inspect v0.0.0-20210209133302-4fd17a0faac2
2022-12-15T22:21:28.771Z [DEBUG] using github.com/hashicorp/terraform-svchost v0.0.0-20200729002733-f050f53b9734
2022-12-15T22:21:28.771Z [DEBUG] using github.com/zclconf/go-cty v1.12.1
2022-12-15T22:21:28.771Z [INFO]  Go runtime version: go1.19.3
2022-12-15T22:21:28.771Z [INFO]  CLI args: []string{"/home/mikael/.tfenv/versions/1.3.6/terraform", "init"}
2022-12-15T22:21:28.771Z [TRACE] Stdout is a terminal of width 148
2022-12-15T22:21:28.771Z [TRACE] Stderr is a terminal of width 148
2022-12-15T22:21:28.771Z [TRACE] Stdin is a terminal
2022-12-15T22:21:28.771Z [DEBUG] Attempting to open CLI config file: /home/mikael/.terraformrc
2022-12-15T22:21:28.771Z [DEBUG] File doesn't exist, but doesn't need to. Ignoring.
2022-12-15T22:21:28.771Z [DEBUG] ignoring non-existing provider search directory terraform.d/plugins
2022-12-15T22:21:28.771Z [DEBUG] ignoring non-existing provider search directory /home/mikael/.terraform.d/plugins
2022-12-15T22:21:28.771Z [DEBUG] ignoring non-existing provider search directory /home/mikael/.local/share/terraform/plugi
2022-12-15T22:21:28.771Z [DEBUG] ignoring non-existing provider search directory /usr/local/share/terraform/plugins
2022-12-15T22:21:28.771Z [DEBUG] ignoring non-existing provider search directory /usr/share/terraform/plugins
2022-12-15T22:21:28.771Z [DEBUG] ignoring non-existing provider search directory /var/lib/snapd/desktop/terraform/plugins
2022-12-15T22:21:28.771Z [INFO]  CLI command args: []string{"init"}
```

Figure 6.34: The Terraform debug output sample

In this screenshot, which is an extract of the execution of Terraform, you can see different information about used providers, Terraform CLI information, and all the steps involved in the execution of Terraform.

There's more...

Instead of having all these traces displayed in the console output, it is also possible to save them in a file.

To do this, just create a second environment variable, `TF_LOG_PATH`, that will contain the path to the log file as a value. Indeed, the logs are often very verbose and difficult to read on the console output. That's why we prefer that the output of the logs is written in a file that can be read more easily.

Moreover, to disable these traces, the `TF_LOG` environment variable must be emptied by assigning it an empty value, as follows:

- **On Windows**: `$env:TF_LOG = ""`
- **On Linux or MacOs**: `unset TF_LOG=""`

See also

- The documentation on the Terraform debug is available here: `https://www.terraform.io/docs/internals/debugging.html`
- Documentation on Terraform's environment variables is available here: `https://www.terraform.io/docs/commands/environment-variables.html`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

7

Sharing Terraform Configuration with Modules

A common problem most developers face in any language is code reuse and how to do it effectively and easily. Hence, the emergence of language, framework, and software packages that are easily reusable in several applications and that can be shared between several teams (such as NuGet, NPM, Bower, PyPI, RubyGems, and many others). In **Infrastructure as Code** (**IaC**) in general, we also encounter the same problems of code structure, its homogenization, and its sharing in the company.

In the *Provisioning infrastructure in multiple environments* recipe in *Chapter 3*, *Scaling Your Infrastructure with Terraform*, we learned about some hierarchies of the Terraform configuration, which gave us a partial answer to the question of how to structure a Terraform configuration well. But that doesn't stop there—Terraform also allows you to create modules with which you can share Terraform configuration between several applications and several teams.

In this chapter, we will study the main stages of the module lifecycle, which are: their creation, the basic use of them and use of them with loops, and also the publishing of them. We will learn about the creation of a Terraform module and its local use, as well as the rapid bootstrapping of the code of a module. We will also study the use of Terraform modules using a public registry or a Git repository. Finally, we will discuss how to use the Terrafile pattern to reference modules' use.

In this chapter, we cover the following recipes:

- Creating a Terraform module and using it locally
- Provisioning multiple instances of a Terraform module

- Using modules from the public registry
- Sharing a Terraform module in the public registry using GitHub
- Using another file inside a custom module
- Using the Terraform module generator
- Generating module documentation
- Using a private Git repository for sharing Terraform modules
- Applying a Terrafile pattern for using modules
- Testing Terraform module code with Terratest

Technical requirements

In this chapter, for some recipes, we will need certain prerequisites, which are as follows:

- **To have Node.js and NPM installed on your computer**: The download website is here: `https://nodejs.org/en/`.
- **To have a GitHub account**: If you don't have one, the creation of an account is free and can be done here: `https://github.com/`.
- **To have an Azure DevOps organization**: You can create one with a Microsoft account or GitHub account here: `https://azure.microsoft.com/en-in/services/devops/`.
- **To have a basic knowledge of Git commands and workflow**: The documentation is available here: `https://git-scm.com/doc`.
- **To know about Docker**: The documentation is here: `https://docs.docker.com/`.
- **To install Golang on our workstation**: The documentation is here: `https://golang.org/doc/install`. We will see the main steps of its installation in the *Testing a Terraform module using Terratest* recipe in *Chapter 11*, *Running Tests and Compliance Security on Terraform Configuration*.

The complete source code for this chapter is available on GitHub at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07`.

Creating a Terraform module and using it locally

A Terraform module is a Terraform configuration that contains one or more Terraform resources.

Once created, this module can be used in several Terraform configuration files either locally or even remotely.

In this recipe, we will look at the steps involved in creating a module and using it locally.

Getting ready

To start this recipe, we will use the Terraform configuration that we already wrote in the *Provisioning infrastructure in multiple environments* recipe of *Chapter 3*, *Scaling Your Infrastructure with Terraform*, and whose sources can be found at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/sample-app`.

The module we will create in this recipe will be in charge of providing an Azure Service Plan, one App Service, and an Application Insights resource in Azure. Its source code is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/Modules/webapp`.

Then, we will write a Terraform configuration that uses this module, and the code is here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/MyApp`.

How to do it…

To create the module, perform the following steps:

1. In a new directory called `moduledemo`, create one subdirectory called `Modules`. Inside the `Modules` directory, create a subdirectory called `webapp`.
2. In the `webapp` subdirectory, create a new `variables.tf` file with the following code:

```
variable "resource_group_name" {
  description = "Resource group name"
}

variable "location" {
  description = "Location of Azure resource"
  default     = "West Europe"
}

variable "service_plan_name" {
  description = "Service plan name"
}

variable "app_name" {
  description = "Name of application"
}
```

3. Then, create the `main.tf` file with the following code:

```
terraform {
  required_version = "~> 1.0"
  required_providers {
    azurerm = {
      version = "~> 3.18"
    }
  }
}

provider "azurerm" {
  features {}
}

resource "azurerm_service_plan" "plan-app" {
  name                = var.service_plan_name
  location            = var.location
  resource_group_name = var.resource_group_name
  os_type   = "Linux"
  sku_name = "B1"
}
resource "random_string" "str" {
  length  = 4
  special = false
  upper   = false
}
resource "azurerm_linux_web_app" "app" {
  name                = "${var.app_name }-${random_string.str.
result}"
  location            = var.location
  resource_group_name = var.resource_group_name
  service_plan_id = azurerm_service_plan.plan-app.id
  site_config {}
}

resource "azurerm_application_insights" "appinsight-app" {
  name                = var.app_name
  location            = var.location
  resource_group_name = var.resource_group_name
  application_type    = "web"
}
```

4. Finally, create the `outputs.tf` file with the following code:

```
output "webapp_id" {
  value = azurerm_linux_web_app.app.id
}
output "webapp_url" {
  value = azurerm_linux_web_app.app.default_hostname
}
```

5. Inside the `moduledemo` folder, create a subfolder called `MyApp`.
6. Inside the `MyApp` subdirectory, create a `main.tf` file with the following code:

```
terraform {
  required_version = "~> 1.0"
  required_providers {
    azurerm = {
      version = "~> 3.18"
    }
  }
}

provider "azurerm" {
  features {}
}
resource "azurerm_resource_group" "rg-app" {
  name     = "RG_MyAPP_demo"
  location = "West Europe"
}

module "webapp" {
  source = "../Modules/webapp"
  service_plan_name = "spmyapp"
  app_name = "myappdemo"
  location = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
}

output "webapp_url" {
  value = module.webapp.webapp_url
}
```

How it works...

In *Step 1*, we create the `moduledemo` directory, which will contain the code for all modules with one subdirectory per module. So, we create a `webapp` subdirectory for our recipe, which will contain the Terraform configuration for the `webapp` module. Then, in *Steps 2*, *3*, and *4*, we create the module code, which is the standard Terraform configuration and contains the following files:

- `main.tf`: This file contains the code of the resources that will be provided by the module.
- `variables.tf`: This file contains the input variables needed by the module.
- `outputs.tf`: This file contains the outputs of the module that can be used in the Terraform configuration that calls this module.

In *Step 5*, we created the `webapp` subfolder, which will contain the Terraform configuration of our application. Finally, in *Step 6*, we created the Terraform configuration of our application with the `main.tf` file.

In the code of this file, we have three blocks:

There is the Terraform `azurerm_resource_group` resource, which provides a resource group.

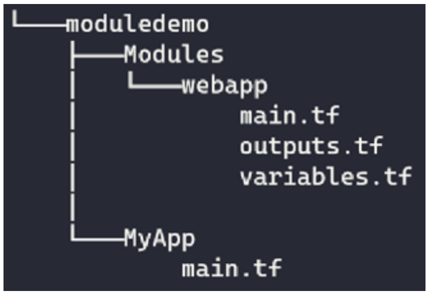
The Terraform configuration that uses the module we created, using the `module "<module name>"` expression. In this module type block, we used the source properties whose values are the relative path of the directory that contains the `webapp` module.

Note that if some variables of the module are defined with default values, then in some cases, it will not be necessary to instantiate them when calling the module.

We also have the Terraform output, `webapp_url`, which gets the output of the module to use as output for our main Terraform configuration.

There's more...

At the end of all of these steps, we obtain the following directory tree:



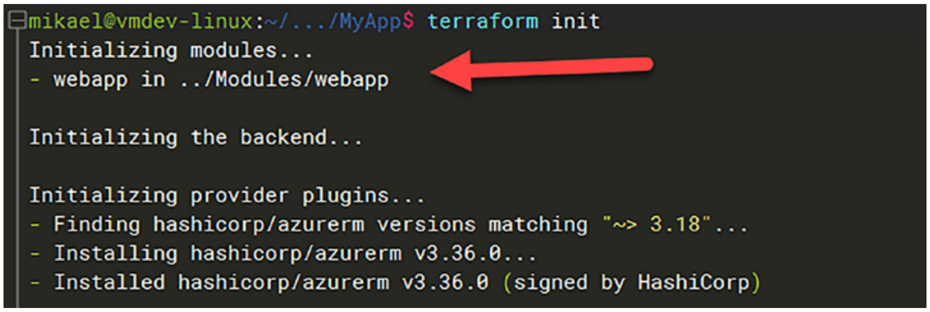
```
└──moduledemo
    ├──Modules
    │   └──webapp
    │          main.tf
    │          outputs.tf
    │          variables.tf
    │
    └──MyApp
           main.tf
```

Figure 7.1: Terraform module structure

To apply this Terraform configuration, you have to navigate in a terminal to the `MyApp` folder containing the Terraform configuration and then execute the following Terraform workflow commands:

```
terraform init
terraform plan -out=app.tfplan
terraform apply app.tfplan
```

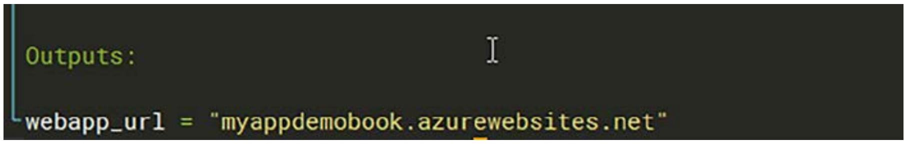
When executing the `terraform init` command, Terraform will get the module's Terraform configuration and hence integrate its configuration with that of the application, as shown in the following screenshot:



```
mikael@vmdev-linux:~/.../MyApp$ terraform init
Initializing modules...
- webapp in ../Modules/webapp

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "~> 3.18"...
- Installing hashicorp/azurerm v3.36.0...
- Installed hashicorp/azurerm v3.36.0 (signed by HashiCorp)
```

Figure 7.2: terraform init get module

Finally, at the end of the execution of the `terraform apply` command, the value of the output is displayed in the terminal, as shown in the following screenshot:



```
Outputs:

webapp_url = "myappdemobook.azurewebsites.net"
```

Figure 7.3: Terraform output from the module

Our Terraform configuration has therefore retrieved the output of the module and used it as the output of our main code.

In this recipe, we have shown the basics of the creation of a Terraform module and its local use. Later in this chapter, we will see how to generate the structure of a module and how to use remote modules in the *Using the Terraform module generator* recipe.

See also

- The documentation on module creation is available at `https://www.terraform.io/docs/modules/index.html`
- General documentation on the modules is available at `https://www.terraform.io/docs/configuration/modules.html`
- Terraform's learning lab on module creation is available at `https://learn.hashicorp.com/terraform/modules/creating-modules`

Provisioning multiple instances of a Terraform module

We learned in *Chapter 3*, *Scaling Your Infrastructure with Terraform*, that some Terraform language features, such as `count` and `for_each`, provision multiple instances of the same resource.

Before the release of version 0.13 of Terraform, it was not possible to create multiple instances of the same module. If we wanted to create several instances of resources that were in a module, we had to apply the `for_each` or `count` expression to each resource that was referenced in the module, which could complicate the maintenance of such a module.

One of the new features of the release of Terraform 0.13 is that we can now use `count` and `for_each` directly on the module block. For more information about this release, read this blog announcement: `https://www.hashicorp.com/blog/announcing-hashicorp-terraform-0-13`.

In this recipe, we will learn how to use `for_each` on Terraform modules.

Let's get started!

Getting ready

To complete this recipe, you'll need to read *Chapter 3*, *Scaling Your Infrastructure with Terraform*, to understand meta-arguments like `count` and `for_each`, and also read the previous recipe of this chapter to understand how to create a Terraform module.

The goal of this recipe is to provision multiple instances of Azure Web Apps in the same Azure Resource Group. The objective of this recipe is to have 1 module with the responsibility to create 1 Web App, and to have a Terraform configuration that calls this module with a loop to create N Web Apps.

To do this Terraform configuration, we will use the `webapp` module we already created, for which the source code is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/Modules/webapp`.

To go ahead in this recipe, we will start with the following Terraform configuration in a new `main.tf` file:

```
resource "azurerm_resource_group" "rg-app" {
  name     = "rg_app_demo_loop"
  location = "West Europe"
}
```

The preceding configuration creates an Azure Resource Group. Now, we will complete this configuration to provision our Azure Web Apps in this Azure Resource Group.

The complete source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/myapp-loop`.

How to do it...

To loop over the Terraform module, perform the following steps:

1. In `main.tf`, add the following Terraform configuration:

   ```
   locals {
     webapp_list = ["webapp12412", "webapp22412"]
   }
   ```

2. Then, add the following configuration

   ```
   module "webapp" {
     source                = "../Modules/webapp"
     for_each              = toset(local.webapp_list)
     app_name              = "${each.key}-${random_string.randomstr.
   result}"
   service_plan_name    = "spmyapp-${each.key}-${random_string.
   randomstr.result}"
     location              = azurerm_resource_group.rg-app.location
   ```

```
    resource_group_name = azurerm_resource_group.rg-app.name
  }
```

3. Finally, to provision these resources, we run the Terraform basic workflow with the `init`, `plan`, and `apply` commands.

How it works...

In *Step 1*, we create a Terraform local expression named `webapp_list`, which contains a list of the two Azure Web Apps that will be provisioned.

Then, in *Step 2*, we call the local `webapp` module and we add a `for_each` meta-argument with the following expression:

```
for_each            = toset(local.webapp_list)
```

By using this `for_each`, the equivalent module will be created "for each" element of the `local.webapp_list` list.

Moreover, to reference the current element of the set during the loop, we use the `each.key` property like this:

```
  app_name              = each.key
  service_plan_name     = "spmyapp-${each.key}"
```

There's more...

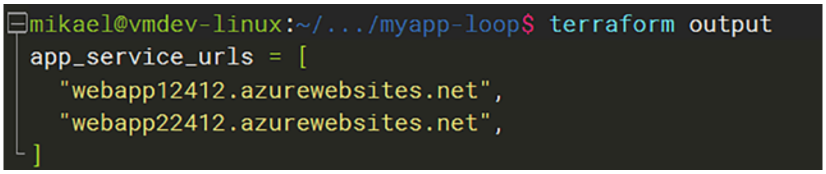
In this recipe, we use the `for_each` expression in the module, we can also use a `count` expression as learned in *Chapter 3*, *Scaling Your Infrastructure with Terraform*.

Additionally, we can also add a Terraform `output` from this module by adding the following Terraform configuration:

```
output "app_service_urls" {
  value = values(module.webapp)[*].webapp_url
}
```

In this output, we get all the Web App URLs created by the module, by using `values(module.webapp)[*]` to get all the output instances in the `webapp` module and filtering to get only the `webapp_url` module outputs.

After executing the `terraform output` command, we get the following result:



```
mikael@vmdev-linux:~/.../myapp-loop$ terraform output
app_service_urls = [
  "webapp12412.azurewebsites.net",
  "webapp22412.azurewebsites.net",
]
```

Figure 7.4: Terraform output from the module using a loop

We can see the hostname of the two provisioned Azure Web Apps.

See also

- The documentation for the `count` and `for_each` in modules is available here: `https://developer.hashicorp.com/terraform/language/modules/develop/refactoring#enabling-count-or-for_each-for-a-module-call`.

Using modules from the public registry

So far, we have studied how to create a module and how to write a Terraform configuration that uses this module locally.

To facilitate the development of Terraform configuration, **HashiCorp** has set up a public Terraform module registry.

This registry solves several problems, such as the following:

- Discoverability with search and filter
- The quality provided by a partner verification process
- Clear and efficient versioning strategy, which is otherwise impossible to solve universally across other existing module sources (**HTTP**, **S3**, and **Git**)

These public modules published in this registry are developed by cloud providers, publishers, communities, or even individual users who wish to share their modules publicly. In this recipe, we will see how to access this registry and how to use a module that has been published in this public registry.

Getting ready

In this recipe, we will write Terraform code from scratch, which does not require any special prerequisites.

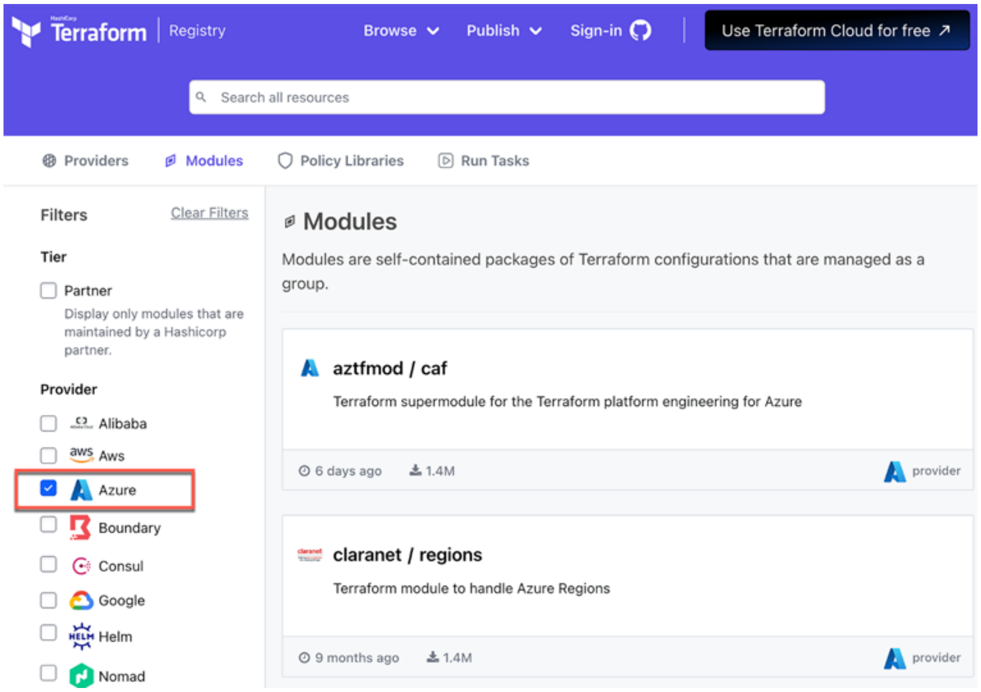
The purpose of this recipe is to provision a Resource Group and network resources in Azure, which are a Virtual Network and subnet. We will see the public module call but we won't look at the Terraform configuration of the module in detail.

The code source for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/publicmodule`.

How to do it…

To use the Terraform module from a public registry, perform the following steps:

1. In a web browser, go to the URL `https://registry.terraform.io/browse/modules`.
2. On this page, in the **Provider** checkbox list in the left panel, select the checkbox with **azurem**:



Modules

Modules are self-contained packages of Terraform configurations that are managed as a group.

aztfmod / caf

Terraform supermodule for the Terraform platform engineering for Azure

6 days ago 1.4M provider

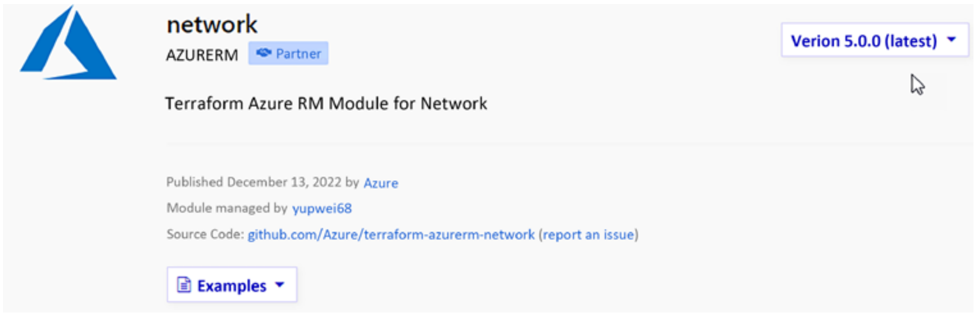
claranet / regions

Terraform module to handle Azure Regions

9 months ago 1.4M provider

Figure 7.5: Terraform Registry provider filter

3. In the results list, click on the item, that is, the **Azure / network** module:



network

AZURERM

Terraform Azure RM Module for Network

Published December 13, 2022 by Azure

Module managed by yupwei68

Source Code: github.com/Azure/terraform-azurerm-network (report an issue)

Figure 7.6: Terraform module registry details

4. Finally, in your workstation, create a new file, `main.tf`, then paste the following code:

```
resource "azurerm_resource_group" "rg" {
  name     = "my-rg"
  location = "West Europe"
}

module "network" {
  source              = "Azure/network/azurerm"
  resource_group_name = azurerm_resource_group.rg.name
  vnet_name           = "vnetdemo"
  address_space       = "10.0.0.0/16"
  subnet_prefixes     = ["10.0.1.0/24"]
  subnet_names        = ["subnetdemo"]
}
```

How it works…

In *Steps 1* to *2*, we explored Terraform's public registry to look for a module that allows the provisioning of resources for Azure (using the `azurerm` filter).

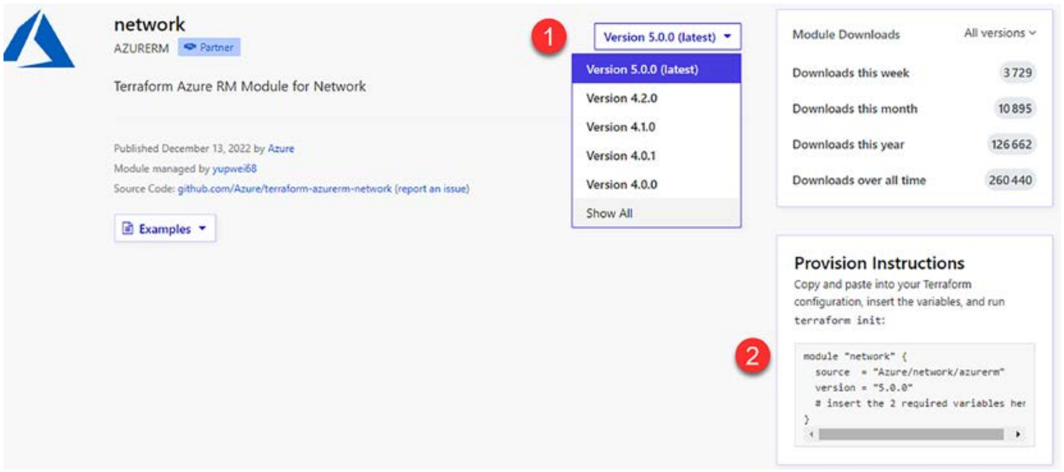
Then, in *Steps 3* and *4*, we accessed the **details** page of the **network** module published by the Azure team.

In *Step 5*, we used this module by specifying the necessary input variables with the `source` property, which has a public module-specific alias, `Azure/network/azurerm`, provided by the registry.

There's more...

We saw in this recipe that using a module from the public registry saves development time. Here, in our recipe, we used a HashiCorp "verified" module (we can read more details about "verified" modules here: `https://developer.hashicorp.com/terraform/registry/modules/verified`), but you can perfectly easily use the other community modules.

It is possible to use the versioning of these modules by choosing the desired version of the module from the version drop-down list and providing provision instructions:


network
AZURERM
Partner
Terraform Azure RM Module for Network
Published December 13, 2022 by Azure
Module managed by yupwei68
Source Code: github.com/Azure/terraform-azurerm-network (report an issue)
Examples
1
Version 5.0.0 (latest)
Version 5.0.0 (latest)
Version 4.2.0
Version 4.1.0
Version 4.0.1
Version 4.0.0
Show All
Module Downloads
All versions
Downloads this week 3 729
Downloads this month 10 895
Downloads this year 126 662
Downloads over all time 260 440
Provision Instructions
2

Copy and paste into your Terraform configuration, insert the variables, and run `terraform init`:

```
module "network" {
  source  = "Azure/network/azurerm"
  version = "5.0.0"
  # insert the 2 required variables her
}
```

Figure 7.8: Terraform module registry version choice

And so in the module call, use the `version` property with the chosen version number.

Note that, like all modules or community packages, you must check that their code is clean and secure before using them by manually reviewing the code inside their GitHub repository. Indeed, in each of these modules, there is a link to the GitHub repository, which contains the source code.

Also, before using a module in a company project, you must consider that if there is a request for the correction or enhancement of a module, you need to create an issue or make a pull request in the GitHub repository of this module. This may require waiting for some time (for the validation waiting time and merge of the pull request) before it can be used with the fix or the requested enhancement.

However, it is worth using these modules, as they are very handy and save a lot of time for demonstrations, labs, and sandbox projects.

We saw the use of the public registry in this recipe; we will study in *Chapter 14*, *Using Terraform Cloud to Improve Collaboration*, how to use a private registry of modules in Terraform.

See also

- The documentation on the Terraform module registry is available at `https://www.terraform.io/docs/registry/`.

Sharing a Terraform module in the public registry using GitHub

In the *Creating a Terraform module and using it locally* recipe of this chapter, we studied how to create a module, and in the previous recipe, *Using a module from the public registry*, of this chapter, how to use a module from the public registry.

In this recipe, we'll see how to publish a module in the public registry by storing its code on GitHub.

Getting ready

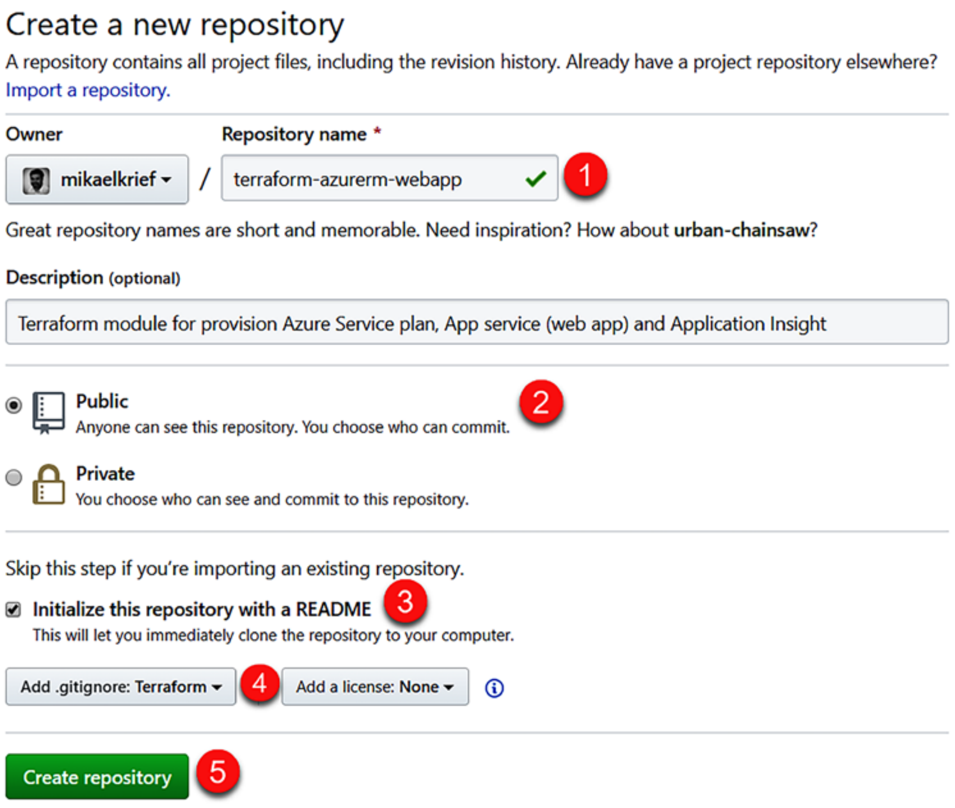
To apply this recipe, we need to have a GitHub account (which is currently the only Git provider available for publishing public modules), which you can create here: `https://github.com/join`. Also, you'll need to know the basics on the Git commands and workflow (`https://www.hostinger.com/tutorials/basic-git-commands`).

Concerning the code of the module we are going to publish, we will use the Terraform configuration of the module we created in the first recipe of this chapter, the source for which is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/Modules/webapp`.

How to do it…

To share our custom module in the public registry, perform the following steps:

1. In our GitHub account, create a new repository named `terraform-azurerm-webapp` with the following basic configuration:



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner: mikaelkrief / Repository name *: terraform-azurerm-webapp

Great repository names are short and memorable. Need inspiration? How about **urban-chainsaw**?

Description (optional): Terraform module for provision Azure Service plan, App service (web app) and Application Insight

[✓ Public] Anyone can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

[✓] Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: Terraform | Add a license: None

Figure 7.9: Create a GitHub repository for the Terraform module

2. In the local workstation, execute the Git command to clone this repository:

```
git clone https://github.com/<your account>/terraform-azurerm-
webapp.git
```

3. Copy the source code from `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/Modules/webapp` and paste it inside the new folder created by the `git clone` command.

4. Update the content of the README.md file with more description of the module role.
5. Stage all files by using the Git commit git add.
6. Commit and push all files in this folder; to perform this action, you can use Visual Studio Code or Git commands (commit and push).
7. Add and push a Git tag v1.0.0 on this commit by executing this command:

```
git tag v1.0.0
git push origin v1.0.0
```

8. In a web browser, go to the URL https://registry.terraform.io/.
9. On this page, click on the **Sign-in** link on the top menu:

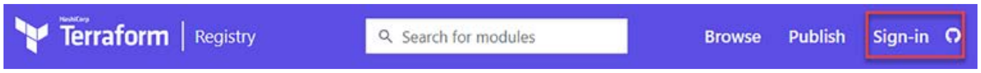


Figure 7.10: Terraform registry sign-in

10. In the newly opened window, click on the **Sign in with GitHub** button, and if prompted, authorize HashiCorp to read your repositories:

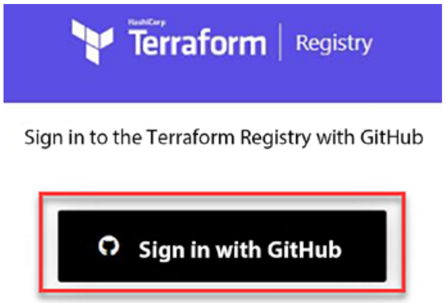


Figure 7.11: Terraform registration with GitHub

11. Once authenticated, click on the **Publish** link on the top menu:

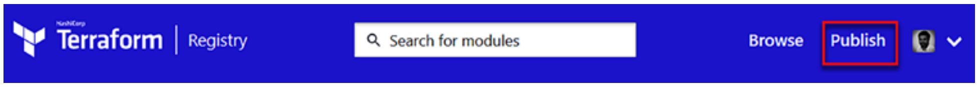


Figure 7.12: Terraform registry Publish button

1. On the next page, select the **<your account>/terraform-azurerm-webapp** repository, which contains the code of the module to publish, and check the **I agree to the Terms of Use** checkbox:

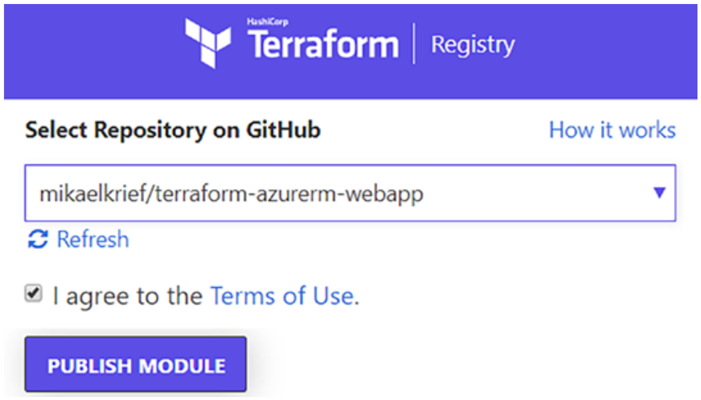

HashiCorp Terraform | Registry
Select Repository on GitHub
How it works
mikaelkrief/terraform-azurerm-webapp
Refresh
I agree to the Terms of Use.
PUBLISH MODULE

Figure 7.13: Terraform module publishing

2. Click on the **PUBLISH MODULE** button and wait for the module page to load.

How it works...

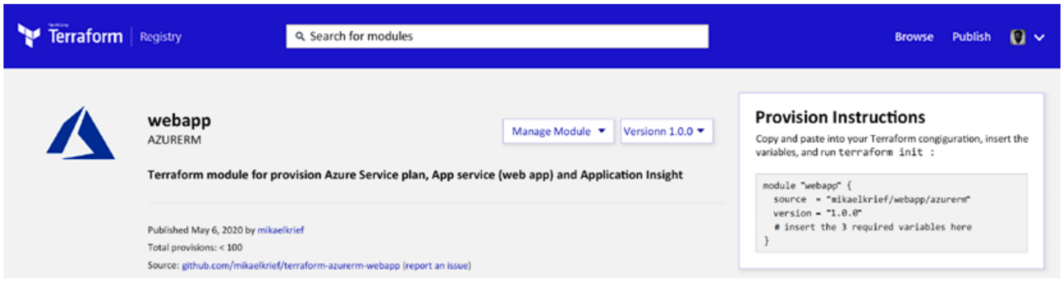
In *Steps 1* and *2*, we created a Git repository in GitHub and cloned it locally, and in *Steps 3* to *6*, we wrote the Terraform configuration for the module (using existing code). We also edited the `README.md` file that will be used as documentation to use the module.

Then, we made a commit and pushed this code in the remote Git repository, and we added a tag, which will be in the form `vX.X.X` and will be used to version the module.

Finally, in *Steps 7* to *12*, we published this module to the public registry by logging in with our GitHub credentials in the registry and then selecting the repository that contains the module code.

The registry automatically detects the version of the module in relation to the Git tag that was pushed (in *Step 6*).

After all of these steps, the module is available in Terraform's public registry, as shown in the following screenshot:


Terraform | Registry
Search for modules
Browse Publish

webapp
AZURERM
Manage Module ▾ Versionn 1.0.0 ▾

Terraform module for provision Azure Service plan, App service (web app) and Application Insight

Published May 6, 2020 by mikaelkrief
Total provisions: < 100
Source: github.com/mikaelkrief/terraform-azurerm-webapp (report an issue)

Provision Instructions
Copy and paste into your Terraform congiguration, insert the variables, and run terraform init :

```
module "webapp" {
  source  = "mikaelkrief/webapp/azurerm"
  version = "1.0.0"
  # insert the 3 required variables here
}
```

Figure 7.14: Terraform module in registry

The module is publicly accessible; the instructions for use are displayed in the right panel and the `README.md` text is displayed as documentation in the content of the page.

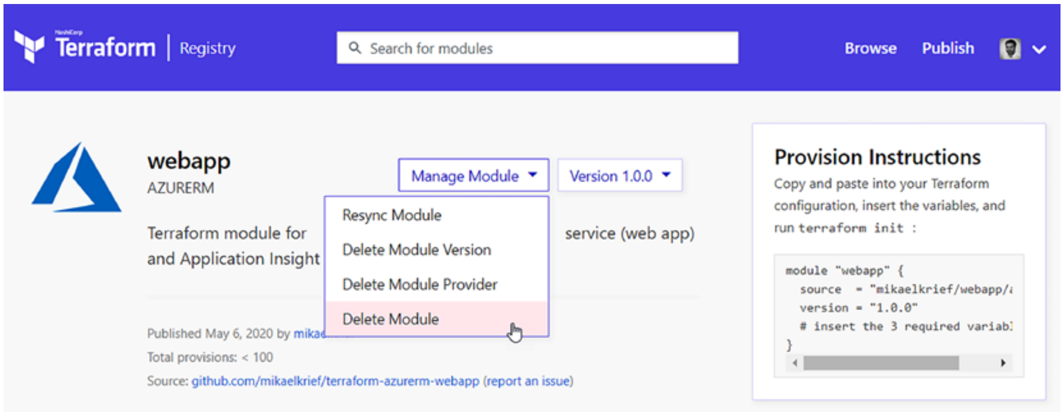
There's more...

Concerning the name of the repository that will contain the module code, it must be composed as follows:

```
terraform-<provider>-<name>
```

Similarly, for the Git tag, it must be in the form `vX.X.X` to be integrated into the registry. To learn more about module requirements, see the documentation: `https://www.terraform.io/docs/registry/modules/publish.html#requirements`.

Once published, it is possible to delete a module by choosing **Delete Module** from the **Manage Module** drop-down list:


Terraform | Registry
Search for modules
Browse Publish

webapp
AZURERM
Manage Module ▾ Version 1.0.0 ▾

Resync Module
Delete Module Version
Delete Module Provider
Delete Module

Terraform module for ... service (web app) and Application Insight

Published May 6, 2020 by mika...
Total provisions: < 100
Source: github.com/mikaelkrief/terraform-azurerm-webapp (report an issue)

Provision Instructions
Copy and paste into your Terraform configuration, insert the variables, and run terraform init :

```
module "webapp" {
  source  = "mikaelkrief/webapp/a
  version = "1.0.0"
  # insert the 3 required variabl
}
```

Figure 7.15: Terraform – Delete Module

Be careful: after deleting it from the registry, the module becomes unusable. Aside from rare cases of experimentation (like here), or possibly some serious security reasons related to leaking IP/secrets, it is more often better to leave modules and all old versions published. This is especially true if the module is public, or generally when it has consumers who depend on it.

See also

- The module publishing documentation is available here: `https://www.terraform.io/docs/registry/modules/publish.html`.
- Documentation on the Registry APIs is available here: `https://www.terraform.io/docs/registry/api.html`.

Using another file inside a custom module

In the *Creating Terraform module and using it locally* recipe of this chapter, we studied the steps to create a basic Terraform module.

We may have scenarios where we need to use another file in the module that does not describe the infrastructure via Terraform (`.tf` extension), for example, in the case where the module needs to execute a script locally.

In this recipe, we will study how to use another file in a Terraform module.

Getting ready

For this recipe, we don't need any prerequisites; we will write the Terraform configuration for the module from scratch.

The goal of this recipe is to create a Terraform module that will execute a Bash script that will perform actions on the local computer (for this recipe, a `hello world` display will suffice).

Since we will be running a Bash script as an example, we will run Terraform under a bash console, like in Linux, WSL for Windows or macOS.

> It is important to keep in mind that provisioners such as this reduce the reusability of your configuration by assuming that the system where Terraform runs has Bash installed. This is otherwise usually not a limiting factor in Terraform as it offers builds for different OSes and architectures and runs cross-platform.

The source code of the created module in this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/Modules/execscript`.

How to do it…

Perform the following steps to use the file inside the module:

1. In a new folder called `execscript` (inside the `Modules` folder), which will contain the code for the module, we create a new file, `script.sh`, with the following content:

   ```
   echo "Hello world"
   ```

2. Create a `main.tf` file in this module and write the following code inside it:

   ```
   resource "null_resource" "execfile" {
     provisioner "local-exec" {
       command = "${path.module}/script.sh"
       interpreter = ["/bin/bash"]
     }
   }
   ```

3. Then, in the Terraform configuration `main.tf` that is new folder `callscript` (in `moduledemo` folder, call this module using the following code:

   ```
   module "execfile" {
     source = "../Modules/execscript"
   }
   ```

   The complete source code of this Terraform configuration is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/callscript`.

4. Finally, in a command-line terminal, navigate to the folder of the Terraform configuration and execute the basic Terraform workflow with the following commands:

   ```
   terraform init
   terraform plan -out="app.tfplan"
   terraform apply app.tfplan
   ```

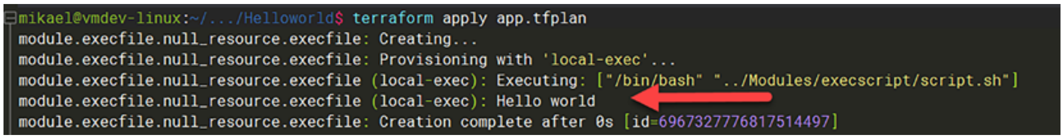
How it works…

In *Steps 1* and *2*, we created a module that executes a script locally using the `local_exec` (`https://www.terraform.io/docs/provisioners/local-exec.html`).

Note that, in this recipe, the `null_resource` Terraform resource very often indicates an attempt at a "hacky solution," a task that is likely better done outside of Terraform entirely. It's available as a "clutch" but if the solution doesn't need a clutch at all, then it's even better.

`local_exec` executes a script in a `script.sh` file that is stored inside the module. To configure the path relative to this `script.sh` file, which can be used during the execution of Terraform, we used the `path.module` expression, which returns the complete path relative to the module.

Then, in *Step 3*, we wrote the Terraform configuration that calls this module. Finally, in *Step 4*, we ran Terraform on this code and we got the following result:



```
mikael@vmdev-linux:~/.../Helloworld$ terraform apply app.tfplan
module.execfile.null_resource.execfile: Creating...
module.execfile.null_resource.execfile: Provisioning with 'local-exec'...
module.execfile.null_resource.execfile (local-exec): Executing: ["/bin/bash" "../Modules/execscript/script.sh"]
module.execfile.null_resource.execfile (local-exec): Hello world
module.execfile.null_resource.execfile: Creation complete after 0s [id=6967327776817514497]
```

Figure 7.16: Terraform exec module with file

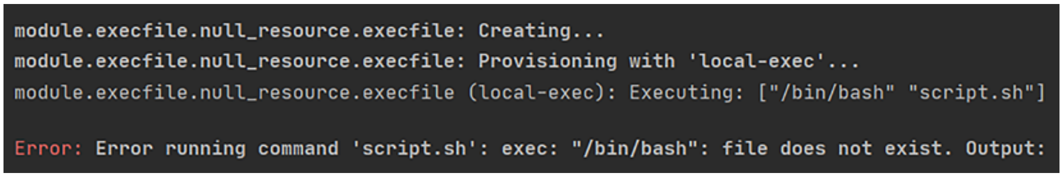
You can see that the script executed successfully and it displays `Hello world` in the console.

There's more...

Let's see what would happen if we hadn't used the `path.module` expression in the code of this module and we had written the module code in the following way:

```
resource "null_resource" "execfile" {
  provisioner "local-exec" {
    command = "script.sh"
    interpreter = ["/bin/bash"]
  }
}
```

When executing the `apply` command, the following error would have occurred:



```
module.execfile.null_resource.execfile: Creating...
module.execfile.null_resource.execfile: Provisioning with 'local-exec'...
module.execfile.null_resource.execfile (local-exec): Executing: ["/bin/bash" "script.sh"]

Error: Error running command 'script.sh': exec: "/bin/bash": file does not exist. Output:
```

Figure 7.17: Terraform error with the bad path.module

Because Terraform interprets the `main.tf` file, it will interpret any relative paths as relative to the current working directory to where `terraform plan/apply` is running from, rather than the module.

See also

- Documentation on the `path.module` expression is available here: `https://developer.hashicorp.com/terraform/language/expressions/references#filesystem-and-workspace-info`.

Using the Terraform module generator

We learned how to create, use, and share a Terraform module and we studied the module file structure best practices, which consist of having a main file, another for variables, and another that contains the outputs of the module. In the *Sharing a Terraform module in the public registry using GitHub* recipe, we also discussed that we could document the use of the module with a `README.md` file.

Apart from these standard files for the operation of the module, we can also add scripts, tests (which we will see in the *Testing Terraform module code with Terratest* recipe), and other files.

For company projects that may need to create a lot of Terraform modules, recreating the structure of the module every time can be a very repetitive and boring task.

To facilitate the creation of the structure of Terraform modules, Microsoft has published a tool that allows us to generate the basic structure (also called a **template**) of a Terraform module.

> A warning, dear reader: the tool we will study in this recipe is no longer maintained by Microsoft. It is therefore recommended to use it if you are a beginner in the creation of Terraform modules just to help you understand the basics and conventions of the structure of a Terraform module.

In this recipe, we will see how to create the base of a module using the module generator.

Getting ready

The software prerequisites for using this module generator are in the following order:

1. Install Node.js (6.0+) locally; its download documentation is available at `https://nodejs.org/en/download/`.

2. Then, install the `npm` package, Yeoman (`https://www.npmjs.com/package/yo`), by executing the following command:

   ```
   npm install -g yo
   ```

To illustrate this recipe, we will use this generator to create the structure of a module that will be in charge of provisioning a Resource Group.

A sample of the generated module from this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/generatedmodule`.

How to do it...

To generate a structure for a Terraform module, perform the following steps:

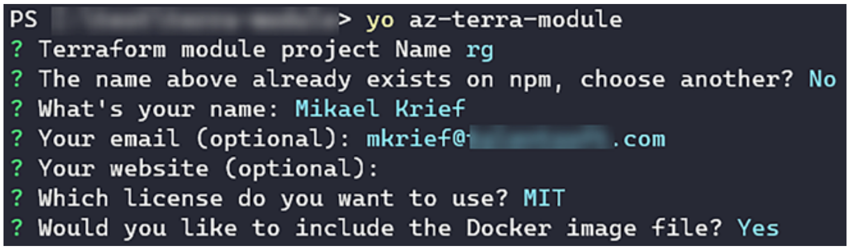
1. In a command-line terminal, execute the following command:

   ```
   npm install -g generator-az-terra-module
   ```

2. Create a new folder with the name of the module, `terraform-azurerm-rg`.
3. Then, in this folder, in the command-line terminal, execute this command:

   ```
   yo az-terra-module
   ```

4. Finally, the generator will ask some questions; type responses like those in the following screenshot:



```
PS [redacted]> yo az-terra-module
? Terraform module project Name rg
? The name above already exists on npm, choose another? No
? What's your name: Mikael Krief
? Your email (optional): mkrief@[redacted].com
? Your website (optional):
? Which license do you want to use? MIT
? Would you like to include the Docker image file? Yes
```

Figure 7.18: Terraform module generator

How it works...

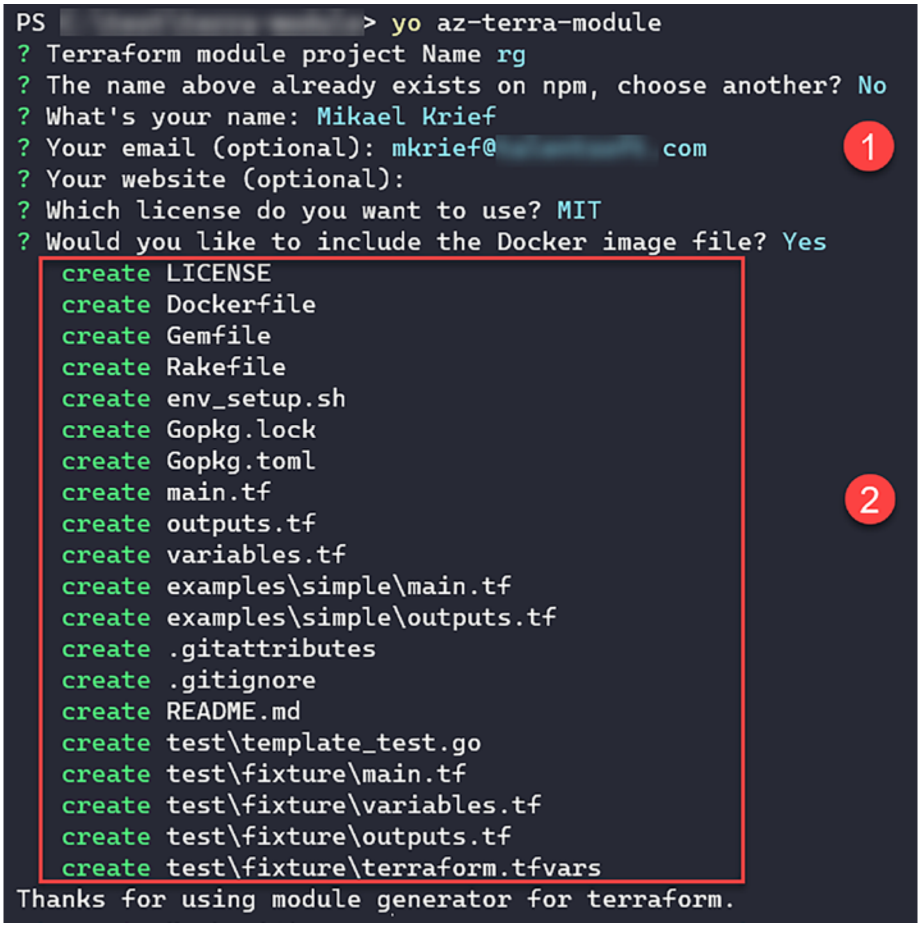
In *Step 1*, we installed the module generator, which is an `npm` package called `generator-az-terra-module`. So we used the classical `npm` command line, which installs a package globally, that is to say, for the whole machine.

In *Step 2*, we created the folder that will contain the code of the module; for our recipe, we used the nomenclature required by Terraform Registry.

In *Steps 3 and 4*, we executed the `az-terra-module` generator. During its execution, this generator asks the user questions that will allow the customization of the module template that will be generated. The first question concerns the name of the module.

The second one concerns the existence of the module in `npm`; we answered `No`. Then, the next three questions concern the module metadata. Finally, the last question asks whether we want to add to the module code a Dockerfile that will be used to run the tests on the module—we answer `Yes`.

At the end of all of these questions, the generator copies all of the files necessary for the module into our directory:

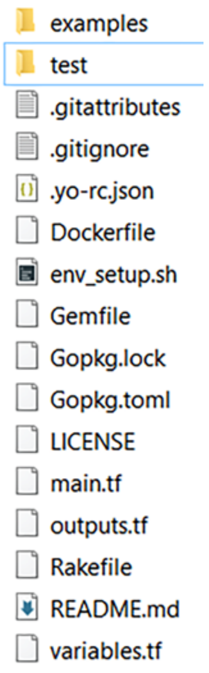


```
PS [redacted]> yo az-terra-module
? Terraform module project Name rg
? The name above already exists on npm, choose another? No
? What's your name: Mikael Krief
? Your email (optional): mkrief@[redacted].com
? Your website (optional):
? Which license do you want to use? MIT
? Would you like to include the Docker image file? Yes
   create LICENSE
   create Dockerfile
   create Gemfile
   create Rakefile
   create env_setup.sh
   create Gopkg.lock
   create Gopkg.toml
   create main.tf
   create outputs.tf
   create variables.tf
   create examples\simple\main.tf
   create examples\simple\outputs.tf
   create .gitattributes
   create .gitignore
   create README.md
   create test\template_test.go
   create test\fixture\main.tf
   create test\fixture\variables.tf
   create test\fixture\outputs.tf
   create test\fixture\terraform.tfvars
Thanks for using module generator for terraform.
```

Figure 7.19: Terraform module generator execution creates files

As you can see on this screen, the generator displays in the terminal the list of folders and files that have been created.

Finally, in File Explorer, we can see all of these files:



examples
test
.gitattributes
.gitignore
.yo-rc.json
Dockerfile
env_setup.sh
Gemfile
Gopkg.lock
Gopkg.toml
LICENSE
main.tf
outputs.tf
Rakefile
README.md
variables.tf

Figure 7.20: Terraform module generator folder structure

The basic structure of our Terraform module is well generated.

There's more...

In this recipe, we saw that it is possible to generate the file structure of a Terraform module. At the end of the execution of this generator, the directory of the created module contains the Terraform files of the module, which will be edited afterward with the code of the module. This folder will also contain other test files and a Dockerfile whose usefulness we will see in the *Testing a Terraform module with Terratest* recipe of *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*.

Also, although this generator is published by Microsoft, it can be used to generate the structure of any Terraform modules you need to create even if it does not provide anything in Azure.

See also

- The source code for the module generator is available on GitHub at `https://github.com/Azure/generator-az-terra-module`.

- Documentation on the use of the generator is available at `https://docs.microsoft.com/en-us/azure/developer/terraform/create-a-base-template-using-yeoman`.
- Yeoman documentation is available at `https://yeoman.io/`.
- The `npm` package of the generator is available at `https://www.npmjs.com/package/generator-az-terra-module`.

Generating module documentation

We learned from the previous recipes that in the composition of a Terraform module, we have input variables, as well as outputs.

As with all packages that are made available to other teams or even publicly, it is very important to document your Terraform module.

The problem with this documentation is that it is tedious to update it for every change in the code and therefore it quickly becomes obsolete.

Among all of the tools in the Terraform toolbox, there is `terraform-docs`, an open source, cross-platform tool that allows the documentation of a Terraform module to be generated automatically.

We will discuss in this recipe how to automatically generate the markdown documentation for a module with `terraform-docs`.

Getting ready

For this recipe, we are going to generate the documentation of the module we created in the *Creating a Terraform module and using it locally* recipe of this chapter, which allowed us to create a web app in Azure, the source for which is here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/Modules/webapp`.

If you are working on a Windows OS, you will need to install **Chocolatey** by following this documentation: `https://chocolatey.org/install`. The documentation we will generate for the `webapp` module is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP07/moduledemo/Modules/webapp/Readme.md`.

How to do it...

Perform the following steps to generate the module documentation:

1. If you are working on a Linux OS, execute the following script in a command-line terminal:

   ```
   curl -L https://github.com/segmentio/terraform-docs/releases/
   download/v0.9.1/terraform-docs-v0.9.1-linux-amd64 -o terraform-docs-
   v0.9.1-linux-amd64
   tar -xf terraform-docs-v0.9.1-linux-amd64
   chmod u+x terraform-docs-v0.9.1-linux-amd64
   sudo mv terraform-docs-v0.9.1-linux-amd64 /usr/local/bin/terraform-
   docs
   ```

   If you are working on a Windows OS, execute the following script:

   ```
   choco install terraform-docs -y
   ```

2. Execute the following script to test the installation:

   ```
   terraform-docs --version
   ```

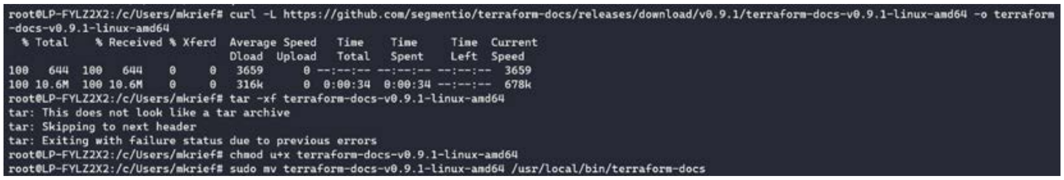
3. In a command-line terminal, navigate into the `moduledemo` folder and execute the following command:

   ```
   terraform-docs markdown Modules/webapp/ > Modules/webapp/Readme.md
   ```

How it works...

In *Step 1*, we install `terraform-docs` according to the operating system. For Linux, the provided script downloads the `terraform-docs` package from GitHub, decompresses it with the TAR tool, gives it execution rights with `chmod`, and finally copies it to the local directory, `/usr/bin/local` (which is already configured in the `PATH` environment variable).

The following screenshot shows the installation in Linux:

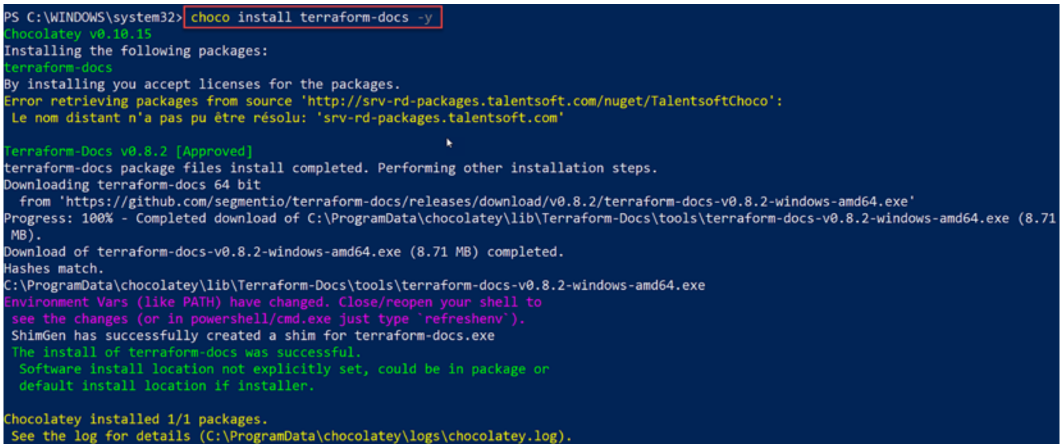


```
root@LP-FYLZ2X2:/c/Users/mkrief# curl -L https://github.com/segmentio/terraform-docs/releases/download/v0.9.1/terraform-docs-v0.9.1-linux-amd64 -o terraform
-docs-v0.9.1-linux-amd64
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   644  100   644    0     0   3659      0 --:--:-- --:--:-- --:--:--  3659
100 10.6M  100 10.6M    0     0   316k      0  0:00:34  0:00:34 --:--:--  678k
root@LP-FYLZ2X2:/c/Users/mkrief# tar -xf terraform-docs-v0.9.1-linux-amd64
tar: This does not look like a tar archive
tar: Skipping to next header
tar: Exiting with failure status due to previous errors
root@LP-FYLZ2X2:/c/Users/mkrief# chmod u+x terraform-docs-v0.9.1-linux-amd64
root@LP-FYLZ2X2:/c/Users/mkrief# sudo mv terraform-docs-v0.9.1-linux-amd64 /usr/local/bin/terraform-docs
```

Figure 7.21: Download terraform-doc on Linux

For Windows, the script uses the `choco install` command from **Chocolatey** to download the `terraform-docs` package.

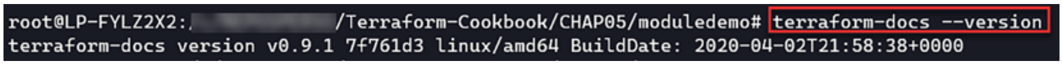
The following screenshot shows the installation in Windows:



```
PS C:\WINDOWS\system32> choco install terraform-docs -y
Chocolatey v0.10.15
Installing the following packages:
terraform-docs
By installing you accept licenses for the packages.
Error retrieving packages from source 'http://srv-rd-packages.talentsoft.com/nuget/TalentsoftChoco':
 Le nom distant n'a pas pu être résolu: 'srv-rd-packages.talentsoft.com'

Terraform-Docs v0.8.2 [Approved]
terraform-docs package files install completed. Performing other installation steps.
Downloading terraform-docs 64 bit
  from 'https://github.com/segmentio/terraform-docs/releases/download/v0.8.2/terraform-docs-v0.8.2-windows-amd64.exe'
Progress: 100% - Completed download of C:\ProgramData\chocolatey\lib\Terraform-Docs\tools\terraform-docs-v0.8.2-windows-amd64.exe (8.71
 MB).
Download of terraform-docs-v0.8.2-windows-amd64.exe (8.71 MB) completed.
Hashes match.
C:\ProgramData\chocolatey\lib\Terraform-Docs\tools\terraform-docs-v0.8.2-windows-amd64.exe
Environment Vars (like PATH) have changed. Close/reopen your shell to
 see the changes (or in powershell/cmd.exe just type `refreshenv`).
 ShimGen has successfully created a shim for terraform-docs.exe
 The install of terraform-docs was successful.
  Software install location not explicitly set, could be in package or
  default install location if installer.

Chocolatey installed 1/1 packages.
 See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
```

Figure 7.22: Download terraform-doc on Windows

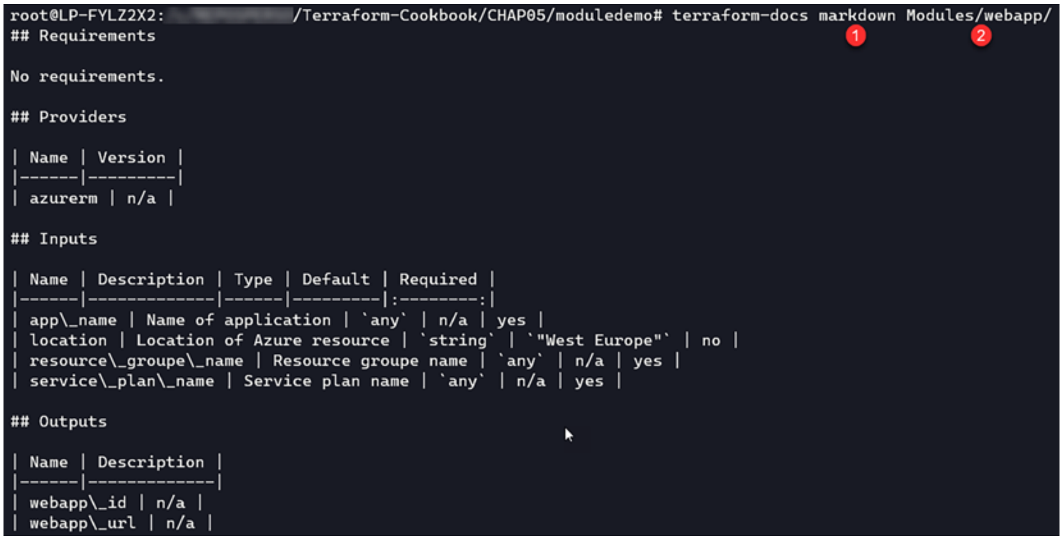
Then, in *Step 2*, we check its installation by running `terraform-docs` and adding the `--version` option. This command displays the installed version of `terraform-docs`, as shown in the following screenshot:



```
root@LP-FYLZ2X2:[redacted]/Terraform-Cookbook/CHAP05/moduledemo# terraform-docs --version
terraform-docs version v0.9.1 7f761d3 linux/amd64 BuildDate: 2020-04-02T21:58:38+0000
```

Figure 7.23: terraform-docs --version command

Finally, in *Step 3*, we execute `terraform-docs` specifying in the first argument the type of format of the documentation. In our case, we want it in `markdown` format. Then, in the second argument, we specify the path of the `modules` directory. At this stage, we could execute the command this way and during its execution, the documentation is displayed in the console, as shown in the following screenshot:



```
root@LP-FYLZ2X2:[redacted]/Terraform-Cookbook/CHAP05/moduledemo# terraform-docs markdown Modules/webapp/
## Requirements

No requirements.

## Providers

| Name | Version |
|------|---------|
| azurerm | n/a |

## Inputs

| Name | Description | Type | Default | Required |
|------|-------------|------|---------|:--------:|
| app\_name | Name of application | `any` | n/a | yes |
| location | Location of Azure resource | `string` | `"West Europe"` | no |
| resource\_groupe\_name | Resource groupe name | `any` | n/a | yes |
| service\_plan\_name | Service plan name | `any` | n/a | yes |

## Outputs

| Name | Description |
|------|-------------|
| webapp\_id | n/a |
| webapp\_url | n/a |
```

Figure 7.24: Generate terraform documentation using terraform-doc

But to go further, we added the `> Modules/webapp/Readme.md` command, which indicates that the content of the generated documentation will be written in the `Readme.md` file that will be created in the module directory.

At the end of the execution of this command, a new `Readme.md` file can be seen inside the module folder that contains the module documentation. The generated documentation is composed of the providers used in the module, the input variables, and the outputs.

There's more...

In our recipe, we chose to generate markdown documentation, but it is also possible to generate it in **JSON**, **XML**, **YAML**, or **text** (**pretty**) format. To do so, you have to add the format option to the `terraform-docs` command. To know more about the available generation formats, read the documentation here: `https://terraform-docs.io/reference/terraform-docs/#subcommands`.

You can also improve your processes by automating the generation of documentation by triggering the execution of `terraform-docs` every time you commit code in Git. For this, you can use a pre-commit Git Hook, as explained in the documentation here: `https://terraform-docs.io/how-to/pre-commit-hooks/`.

Also, to get the latest version of `terraform-docs`, follow the release here – `https://github.com/segmentio/terraform-docs/releases`.

> If you want to publish your module in the Terraform registry as we have seen in the *Sharing a Terraform module in the public registry using GitHub* recipe in this chapter, you do not need to generate this documentation because it is already included in the registry's functionalities.

See also

- The source code for `terraform-docs` is available here: `https://github.com/segmentio/terraform-docs`.
- The Chocolatey `terraform-docs` package page is available here: `https://chocolatey.org/packages/Terraform-Docs`.

Using a private Git repository for sharing a Terraform module

In this chapter dedicated to Terraform modules, we have seen that it is possible to put the code of a module in a GitHub repository to publish it in the Terraform public registry.

However, in companies of any size, there is a need to create modules without exposing the code of these modules publicly in GitHub repositories, that is, making it accessible to everyone.

What you need to know is that there are several types of Terraform module sources, as indicated in this documentation: `https://www.terraform.io/docs/modules/sources.html`.

In this recipe, we will study how to expose a Terraform module through a private Git repository. Either this Git server is installed internally (so-called on-premises) or on the cloud, i.e., SaaS, but requires authentication to access the repository.

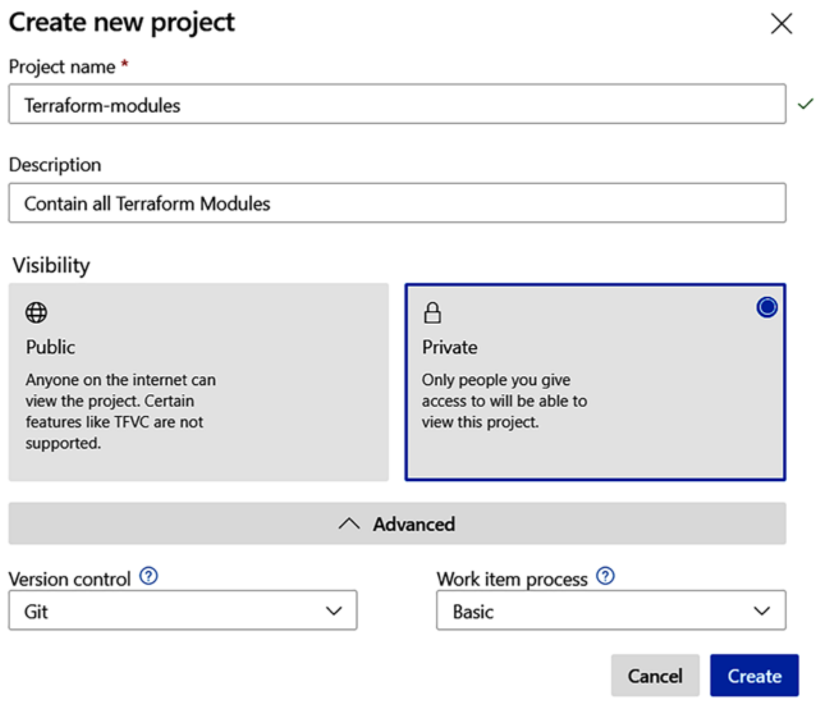
Getting ready

For this recipe, we will use a Git repository in **Azure Repos** (**Azure DevOps**), which is free and requires authentication to access it. For more information and how to create a free Azure DevOps account, go to `https://azure.microsoft.com/en-us/services/devops/`.

We will use an SSH key to clone repositories in Azure DevOps, and more information on how to configure SSH keys can be found here: `https://learn.microsoft.com/en-us/azure/devops/repos/git/use-ssh-keys-to-authenticate?view=azure-devops`.

As a prerequisite, we need a project that has already been created; it can be named, for example, `Terraform-modules`, and it will contain the Git repository of all of the modules.

The next screenshot shows the form for creating this Azure DevOps project:



Create new project

Project name *: Terraform-modules

Description: Contain all Terraform Modules

Visibility

Public — Anyone on the internet can view the project. Certain features like TFVC are not supported.

[✓ Private — Only people you give access to will be able to view this project.]

Advanced

Version control: Git

Work item process: Basic

Figure 7.25: Azure DevOps creating a private project

The purpose of this recipe is not to focus on the use of Azure DevOps; we will use it just to have an example of a private repository.

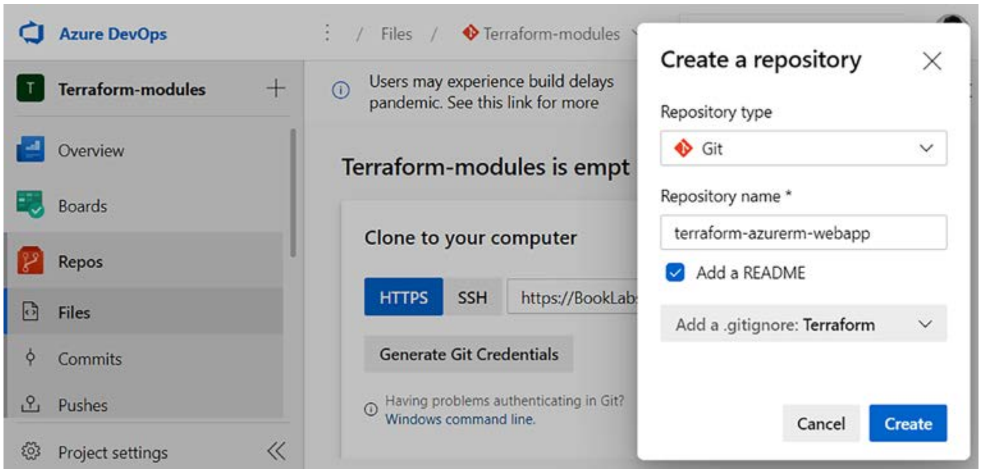
You will also need to know the basics of the commands and workflow of Git: `https://www.hostinger.com/tutorials/basic-git-commands`.

Concerning the code of the module that we are going to put in Azure Repos, we are going to use the code of the module that we created in the first recipe of this chapter, the source code for which is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/Modules/webapp`.

How to do it...

To use a private module repository, we need to perform the following steps:

1. In the Azure DevOps project, `Terraform-modules`, create a new Git repository named `terraform-azurerm-webapp` with basic configuration, as shown in the following screenshot:



Users may experience build delays pandemic. See this link for more

Terraform-modules is empt

Clone to your computer

Having problems authenticating in Git? Windows command line.

Create a repository

Repository type: Git

Repository name *: terraform-azurerm-webapp

✓ Add a README

Add a .gitignore: Terraform

Figure 7.26: Azure Repos | Create a repository

2. In a local workstation, execute the Git command for cloning this repository using HTTPS (and replace <organisation> with your Azure DevOps organisation):

```
git clone git@https://<your azdo organisation>@dev.azure.com/
BookLabs/Terraform-modules/_git/terraform-azurerm-webapp
```

3. During the first operation, you will have to enter your Azure DevOps login and password for identification.
4. Copy the source code from `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/moduledemo/Modules/webapp` and paste it into the new folder created by the `git clone` command.

5. Update the content of the `Readme.md` file with a description of the module's purpose.
6. Commit and push all files into this folder; to perform this action, you can use VS Code or Git commands:

```
git add .
git commit -m "add code"
git push origin master
```

7. Add and push a Git tag `v1.0.0` on this commit by executing this command:

```
git tag v1.0.0
git push origin v1.0.0
```

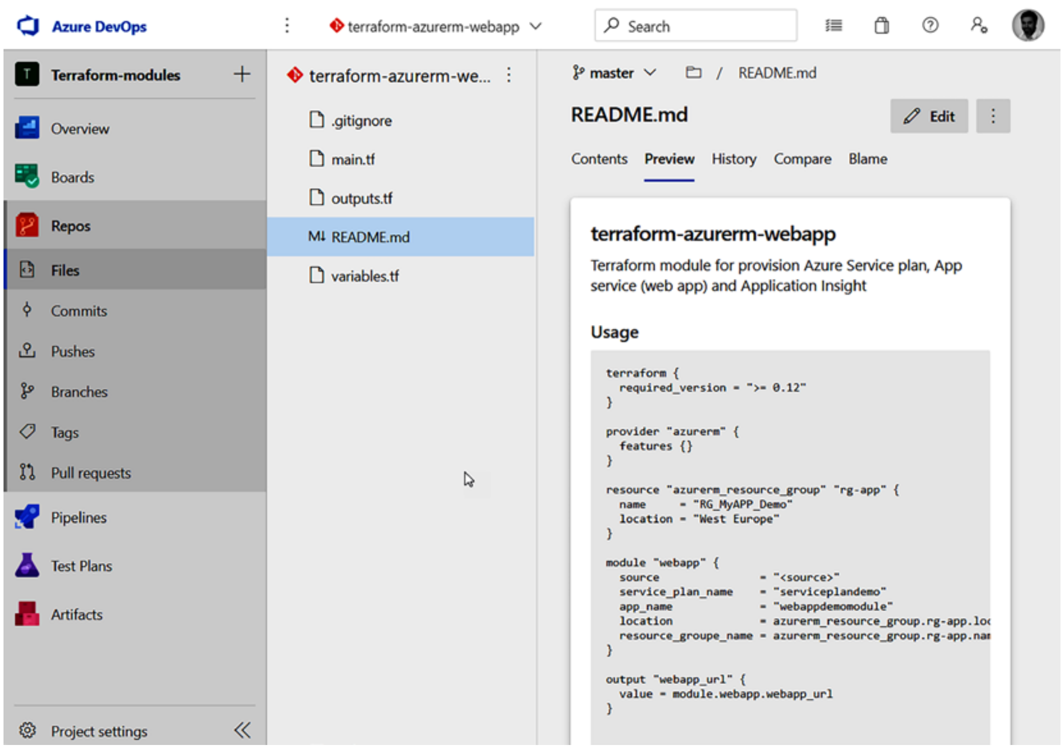
This operation can also be done via the web interface of Azure Repos, in the **Tags** tab of the repository.

8. Finally, in the Terraform `main.tf` file, the following code is written, which uses the module:

```
resource "azurerm_resource_group" "rg-app" {
  name     = "RG_MyAPP_Demo2"
  location = "West Europe"
}

module "webapp" {
  source              = "git@ssh.dev.azure.com:v3/BookLabs/
Terraform-modules/terraform-azurerm-webapp?ref=v1.0.0"
  service_plan_name   = "spmyapp2"
  app_name            = "myappdemobook2"
  location            = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
}
output "webapp_url" {
  value = module.webapp.webapp_url
}
```

How it works...

In *Steps 1* and *2*, we created a Git repository in Azure Repos and cloned it locally. Then, in *Steps 3* to *7*, we wrote the Terraform configuration for the module (using already existing code). We also edited the `Readme.md` file, which will be used as documentation for the use of the module. Then, we made a commit and pushed this code into the remote Git repository. The following screenshot shows the remote repository in Azure Repos:



terraform-azurerm-webapp

Terraform module for provision Azure Service plan, App service (web app) and Application Insight

Usage

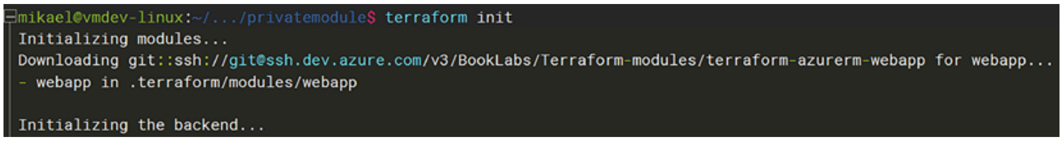
```
terraform {
  required_version = ">= 0.12"
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg-app" {
  name     = "RG_MyAPP_Demo"
  location = "West Europe"
}

module "webapp" {
  source              = "<source>"
  service_plan_name   = "serviceplandemo"
  app_name            = "webappdemomodule"
  location            = azurerm_resource_group.rg-app.loc
  resource_groupe_name = azurerm_resource_group.rg-app.nam
}

output "webapp_url" {
  value = module.webapp.webapp_url
}
```

Figure 7.27: Azure Repos module README.md

Then, we added a Git tag, which will be in the format `vX.X.X` and will be used to version the module.

Finally, in *Step 8*, we wrote the Terraform configuration, which remotely uses this module with a Git-type source. For this, we specified the `source` property of the module with the Git URL of the repository. In addition to this URL, we added the `ref` parameter, to which we give the Git tag we created as a value.

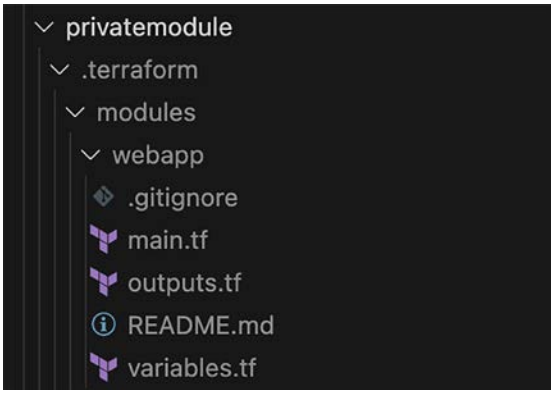
It will be during the execution of the `terraform init` command that Terraform will clone the repository locally:



```
mikael@vmdev-linux:~/.../privatemodule$ terraform init
Initializing modules...
Downloading git::ssh://git@ssh.dev.azure.com/v3/BookLabs/Terraform-modules/terraform-azurerm-webapp for webapp...
- webapp in .terraform/modules/webapp

Initializing the backend...
```

Figure 7.28: terraform init downloads the private module

The module code will be cloned into the Terraform context directory, as shown in the following screenshot:



```
privatemodule
  .terraform
    modules
      webapp
        .gitignore
        main.tf
        outputs.tf
        README.md
        variables.tf
```

Figure 7.29: Terraform module download during init

The `webapp` module is downloaded inside the `.terraform` folder.

There's more...

In this recipe, most of the steps are identical to the ones already studied in the *Sharing a Terraform module in the public registry using GitHub* recipe, in which we stored the module code in GitHub and shared it in the public registry. The difference is that, in *Step 8* of this recipe, we filled the value of the `source` property with the Git repository URL.

The advantages of using a private Git repository are, on the one hand, that it's only accessible to people who have permission for that repository. On the other hand, in the `ref` parameter that we put in the module call URL, we used a specific version of the module using a Git tag. We can also name a specific Git branch, which is very useful when we want to evolve the module without impacting the main branch.

Note that branches should only be used during development, if necessary – since the underlying branch can change at any time and break things. Production code should always point to a tag, which is treated as immutable.

We could also very well store the module's code in a GitHub repository and fill the `source` properties with the GitHub repository URL, as shown in this documentation: `https://www.terraform.io/docs/modules/sources.html#github`.

In this recipe, we took a Git repository in Azure DevOps as an example, but it also works very well with other Git repository providers such as Bitbucket (`https://www.terraform.io/docs/modules/sources.html#bitbucket`).

Regarding Git repository authentication, you can check out this documentation at `https://www.terraform.io/docs/modules/sources.html#generic-git-repository` for information on access to and the authentication of a Git repository in HTTPS or SSH.

See also

- The module source documentation is available here: `https://www.terraform.io/docs/modules/sources.html`.

Applying a Terrafile pattern for using modules

We saw throughout this chapter's recipes how to create Terraform modules and how to use them either locally or remotely with a public registry or Git repositories.

However, when you have a Terraform configuration that uses many modules, managing these modules can become complicated. This is indeed the case when the versions of these modules change; it is necessary to browse through all of the Terraform configurations to make version changes. Moreover, we do not have global visibility of all of the modules called in this Terraform configuration, as well as their versions.

Analogous to the classic package managers (**NPM** and **NuGet**), a pattern has emerged that allows users to gather the configuration of the Terraform modules used in a Terraform configuration in a centralized file called a **Terrafile**.

In this recipe, we will study how to use the Terrafile pattern to manage the sources of the Terraform modules.

Getting ready

For this recipe, we will use the Terraform source code that is already written and available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP07/terrafile/initial/main.tf`. This Terraform configuration is at first classically configured—it calls several modules and in each of the calls to these modules, we use the source property with the GitHub repositories' URLs.

Moreover, we will execute a code written in Ruby with Rake (`https://github.com/ruby/rake`). For this, we need to have Ruby installed on our computer. The installation documentation is available here: `https://www.ruby-lang.org/en/documentation/installation/`. However, no prior Ruby knowledge is required; the complete script is provided in the source code for this recipe.

The goal of this recipe will be to integrate the Terrafile pattern into this code by centralizing the management of the modules to be used.

The code source of this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP07/terrafile`.

How to do it...

Perform the following steps to use the Terrafile pattern:

1. Copy the content from `main.tf`, available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP07/terrafile/initial/main.tf`, into a new folder.
2. In this new folder, create a new file called `Terrafile` (without an extension) with the following content:

```
rg-master:
  source:  "https://github.com/mikaelkrief/terraform-azurerm-resource-group.git"
  version: "master"
webapp-1.0.0:
  source:  "https://github.com/mikaelkrief/terraform-azurerm-webapp.git"
  version: "1.0.0"
network-3.0.1:
  source:  "https://github.com/Azure/terraform-azurerm-network.git"
  version: "v3.0.1"
```

3. Create another new file, `Rakefile` (without an extension), with the following content (the complete source code is at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP07/terrafile/new/Rakefile`):

```
.....
desc 'Fetch the Terraform modules listed in the Terrafile'
task :get_modules do
  terrafile = read_terrafile
  create_modules_directory
  delete_cached_terraform_modules
  terrafile.each do |module_name, repository_details|
    source  = repository_details['source']
    version = repository_details['version']
    puts "[*] Checking out #{version} of #{source}
...".colorize(:green)
    Dir.mkdir(modules_path) unless Dir.exist?(modules_path)
    Dir.chdir(modules_path) do
       #puts "git clone -b #{version} #{source} #{module_name} &> /
dev/null".colorize(:green)
      'git clone -q -b #{version} #{source} #{module_name}'
    end
  end
end
```

4. In `main.tf`, update all the `source module` properties with the following content:

```
module "resourcegroup" {
  source = "./modules/rg-master"
 ...
}

module "webapp" {
  source = "./modules/webapp-1.0.0"
...
}

module "network" {
  source = "./modules/network-3.0.1"
...
}
```

5. In the command-line terminal, inside this folder, execute the following script:

```
gem install colorize
rake get_modules
```

6. Finally, run the basic Terraform workflow executing the commands `terraform init`, `plan`, and `apply`.

How it works...

The mechanism of the Terrafile pattern is that instead of using the Git sources directly in module calls, we reference them in a file in Terrafile YAML format. In the Terraform configuration, in the module call, we instead use a local path relative to the `modules` folder. Finally, before executing the Terraform workflow, we execute a script that runs through this `Terrafile` file and will locally clone each of these modules referenced in its specific folder (which is in the module call).

In *Step 1*, we created the `Terrafile` file, which is in YAML format and contains the repository of the modules we are going to use in the Terraform configuration. For each of the modules, we indicate the following:

- The name of the folder where the module will be copied
- The URL of the Git repository of the module
- Its version, which is the Git tag or its branch

In *Step 2*, we wrote the Ruby Rake script called Rakefile, which, when executed, will interpret the Terrafile and will execute the `git clone` command on all modules into the specified folders.

Then, in *Step 3*, we modify the Terraform configuration to call the modules, no longer with the Git URL but with the relative path of their specified folder in the Terrafile.

Finally, in *Step 4*, we execute the Rakefile script by calling the `get_modules` function of this script, which will make a Git clone of all of these modules in their folders.

Once these steps are done, we can execute the classic Terraform workflow commands with `init`, `plan`, and `apply`.

There's more...

As we learned in this recipe, we have a `Terrafile` file that serves as a source of truth for the modules we will use in our Terraform configuration. This allows for better management and maintenance of the modules and versions to be used.

In this file, for each module, we have specified its destination folder, and as you can see, we have added the version number in the folder name. Hence, the name of the folder is unique and will allow us to use several versions of the same module in the Terraform configuration. The following code shows an extract of a Terrafile with two different versions of the same module:

```
network-3.0.1:
  source:  "https://github.com/Azure/terraform-azurerm-network.git"
  version: "v3.0.1"
network-2.0.0:
  source:  "https://github.com/Azure/terraform-azurerm-network.git"
  version: "v2.0.0"
```

Also, it allows you to specify, if necessary, the authorized Git credentials to clone the module code. Be careful, however, not to write passwords in this file, which will be archived in a Git repository.

In this recipe, the Rakefile script was provided and is available in the original article on the Terrafile pattern (`https://bensnape.com/2016/01/14/terraform-design-patterns-the-terrafile/`). You are free to adapt it according to your needs.

The essentials of the Terrafile pattern are not the script, the language, or the format used but, rather, its working principle. There are alternative scripts and tools to use this Rakefile with, for example, a Python script available at `https://github.com/claranet/python-terrafile`, or a tool written in Go available at `https://github.com/coretech/terrafile`.

Finally, there is another similar solution for referencing private Terraform modules inside your Terraform configuration – by using Git submodules (`https://git-scm.com/book/en/v2/Git-Tools-Submodules`), or the the **vendir** tool from Carvel, available here: `https://carvel.dev/vendir/`.

See also

- The main reference article on the Terrafile pattern is available here: `https://bensnape.com/2016/01/14/terraform-design-patterns-the-terrafile/`.
- The Python Terrafile package is available here: `https://pypi.org/project/terrafile/` and its use is described here `https://github.com/claranet/python-terrafile`.
- The Terrafile tool written in Go is available here: `https://github.com/coretech/terrafile`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

8

Provisioning Azure Infrastructure with Terraform

Terraform contains a multitude of providers that enable the provisioning of various types of infrastructure, whether in the cloud or an on-premises data center.

In the previous chapters of this book, we studied the basic concepts of the Terraform language, as well as the Terraform command-line interface, and we saw the sharing of Terraform configuration using modules. In addition, even if the Terraform configuration examples mentioned in the previous chapters are based on the provider's **azurerm**, all the recipes we have seen in the previous chapters are generic and can be used by all Terraform providers.

In this chapter, we will focus on using Terraform to provision a cloud infrastructure in Azure. We will start with its integration into Azure Cloud Shell, its secure authentication, and the protection of the Terraform state file in an Azure storage account.

You will learn how to run **Azure Resource Manager** (**ARM**) templates and Azure CLI scripts with Terraform and how to retrieve the Azure resource list with Terraform. Then we'll look at how to protect sensitive data in Azure Key Vault using Terraform. We will write two case studies in Azure, with the first showing the provisioning and configuration of an IaaS infrastructure consisting of VMs, and the second showing the provisioning of a PaaS infrastructure in Azure. Finally, we will go further with the cost estimation of Azure resources based on Terraform configuration and the use of a new `AzAPI` provider for the generation of Terraform configuration from an already existing infrastructure.

In this chapter, we will cover the following recipes:

- Using Terraform in Azure Cloud Shell
- Protecting the Azure credential provider
- Protecting the state file in the Azure remote backend
- Executing ARM templates in Terraform
- Executing Azure CLI commands in Terraform
- Using Azure Key Vault with Terraform to protect secrets
- Provisioning and configuring an Azure VM with Terraform
- Building Azure serverless infrastructure with Terraform
- Generating a Terraform configuration for existing Azure infrastructure
- Enabling optional Azure features
- Estimating Azure cost of infrastructure using `Infracost`
- Using the `AzAPI` Terraform provider

Technical requirements

To apply the recipes in this chapter, you must have an Azure subscription. If you don't have one, you can create an Azure account for free at this site: `https://azure.microsoft.com/free/`.

For the Terraform Azure provider to provision and manipulate resources in Azure, the provider must authenticate in Azure using an Azure account, and that account must have the correct authorizations. Read this documentation for more details: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/service_principal_client_certificate`.

The complete source code for this chapter is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08`.

Using Terraform in Azure Cloud Shell

In *Chapter 1*, *Setting Up the Terraform Environment*, we studied the steps involved in installing Terraform on a local machine.

In Azure Cloud Shell, Microsoft has integrated Terraform into the list of tools that are installed by default.

In this recipe, we will see how to write a Terraform configuration and use Terraform in Azure Cloud Shell.

Getting ready

The prerequisite for this recipe is to have an Azure subscription and to be connected to this subscription via the Azure portal, which is accessible here: `https://portal.azure.com/`.

Note that this prerequisite applies to all recipes in this chapter.

In addition, you need to associate your Azure Cloud Shell with an existing Azure Storage Account or create a new one, as explained in the following documentation: `https://docs.microsoft.com/azure/cloud-shell/persisting-shell-storage`.

How to do it...

To use Terraform in Azure Cloud Shell, perform the following steps:

1. In the Azure portal, open Azure Cloud Shell by clicking the Azure Cloud Shell button in the top menu, as shown in the following screenshot:

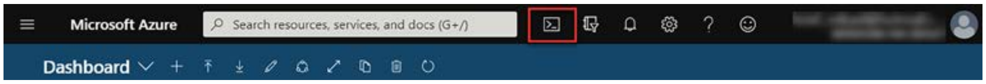
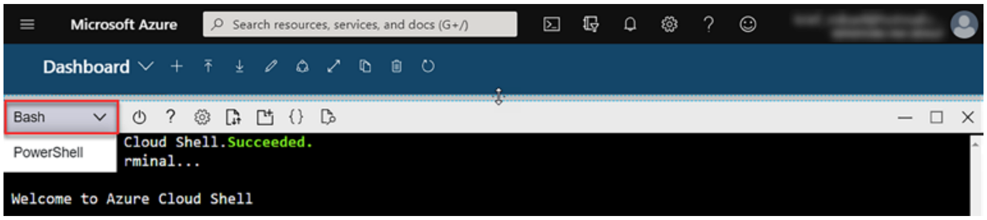


Figure 8.1: Azure Cloud Shell button

2. Inside the Cloud Shell panel, in the top menu, in the dropdown, choose `Bash` mode:



```
Cloud Shell.Succeeded.
rminal...

Welcome to Azure Cloud Shell
```

Figure 8.2: Azure Cloud Shell Bash mode

3. In the Cloud Shell terminal, create a new folder, `demotf`, inside the default `clouddrive` folder by executing the following command:

```
mkdir clouddrive/demotf
```

Inside this new folder, enter the `cd clouddrive/demotf` command.

4. To write a Terraform configuration inside a web IDE called the Azure Cloud Shell editor, execute the `code` command. For more information, read the documentation here: `https://learn.microsoft.com/en-us/azure/cloud-shell/using-cloud-shell-editor`.
5. In the Azure Cloud Shell editor, write the following sample Terraform configuration:

```
terraform {
  required_version = "~> 1.0"
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg-app" {
  name     = "RG-TEST-DEMO"
  location = "westeurope"
}
```

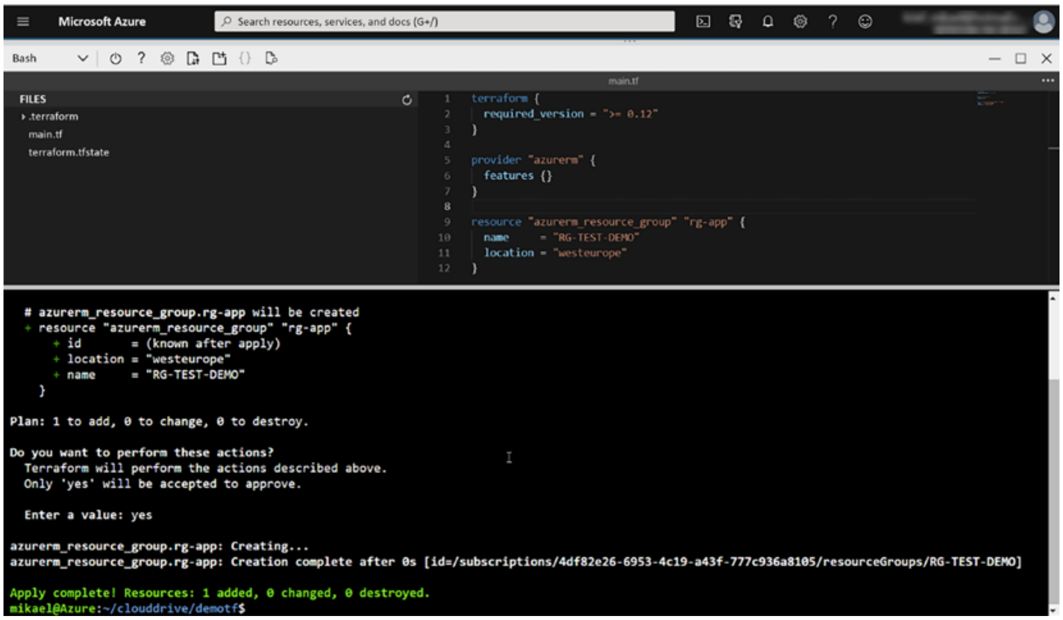
6. Save this file by using the *Ctrl + S* shortcut and name this file `main.tf`.
7. Finally, to apply this Terraform configuration to the Cloud Shell terminal, execute the Terraform workflow as follows:

```
terraform init
terraform plan -out=app.tfplan
terraform apply app.tfplan
```

How it works...

In this recipe, we used the integrated environment of Azure Cloud Shell, which consists of a command-line terminal that we chose to use in Bash mode. In addition, in *Steps 5* and *6*, we used the built-in editor, using the `code` command, to write a Terraform configuration, which also has syntax highlighting for Terraform files. And finally, in *Step 7*, we used the Terraform CLI, which is already installed in this Cloud Shell environment to provision our infrastructure with the execution of the Terraform workflow commands.

The following screenshot shows Azure Cloud Shell with Terraform execution:



```
terraform {
  required_version = ">= 0.12"
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg-app" {
  name     = "RG-TEST-DEMO"
  location = "westeurope"
}
```

```
  # azurerm_resource_group.rg-app will be created
  + resource "azurerm_resource_group" "rg-app" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "RG-TEST-DEMO"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

azurerm_resource_group.rg-app: Creating...
azurerm_resource_group.rg-app: Creation complete after 0s [id=/subscriptions/4df82e26-6953-4c19-a43f-777c936a8105/resourceGroups/RG-TEST-DEMO]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
mikael@Azure:~/clouddrive/demotf$
```

Figure 8.3: Terraform on Azure Cloud Shell

We can see in the preceding screenshot the top panel with the integrated editor and, in the bottom panel, the command line with the execution of the Terraform command.

There's more...

For this recipe, we chose the option to edit the Terraform files directly in the editor, which is in Cloud Shell, but the following alternative options are available:

- The Terraform files could be created and edited using the **Vim** tool (Linux editor: `https://www.linux.com/training-tutorials/vim-101-beginners-guide-vim/`), which is built into Cloud Shell.
- We could also have edited the Terraform files locally on our machine and then copied them to the Azure Storage service that is connected to Azure Cloud Shell.
- If the files are stored in a Git repository, we could also have cloned the repository directly into the Cloud Shell storage by running a `git clone` command in the Cloud Shell command-line terminal.

Also, regarding Terraform's authentication to perform actions in Azure, we did not need to take any action because Azure Cloud Shell allows direct authentication to our active Azure subscription, and Terraform, which is in Cloud Shell, automatically inherits that authentication.

On the other hand, if you have several subscriptions, prior to executing the Terraform workflow, you must choose the subscription target by executing the following command:

```
az account set -s <subscription _id>
```

This chosen subscription then becomes the default subscription during execution. Refer to the documentation at `https://docs.microsoft.com/cli/azure/account?view=azure-cli-latest - az-account-set`.

Regarding the version of Terraform that is installed on Cloud Shell, you can check it by running the `terraform --version` command. You need to check that your Terraform configuration is compatible with this version before executing.

Finally, as regards the recommended use of Azure Cloud Shell for Terraform, it can only be used for development and testing. It cannot be integrated into a CI/CD pipeline using your personal permissions on Azure to provision resources. For this reason, in the next recipe, we will look at how to securely authenticate Terraform to Azure.

See also

- Refer to this blog post, which also shows the use of Terraform in Azure Cloud Shell
- Documentation that explains the use of Azure Cloud Shell: `https://docs.microsoft.com/azure/cloud-shell/using-cloud-shell-editor`
- A tutorial that shows how to use and configure locally installed Visual Studio Code to execute a Terraform configuration in Azure Cloud Shell: `https://docs.microsoft.com/azure/developer/terraform/configure-vs-code-extension-for-terraform`

Protecting the Azure credential provider

In the previous recipe, we studied how to automatically authenticate the Terraform context in Azure Cloud Shell with our personal account and permissions. However, in a company context, as well as in production, it is very bad practice to use your personal account as this could expire, be deleted, or, even worse, be misused.

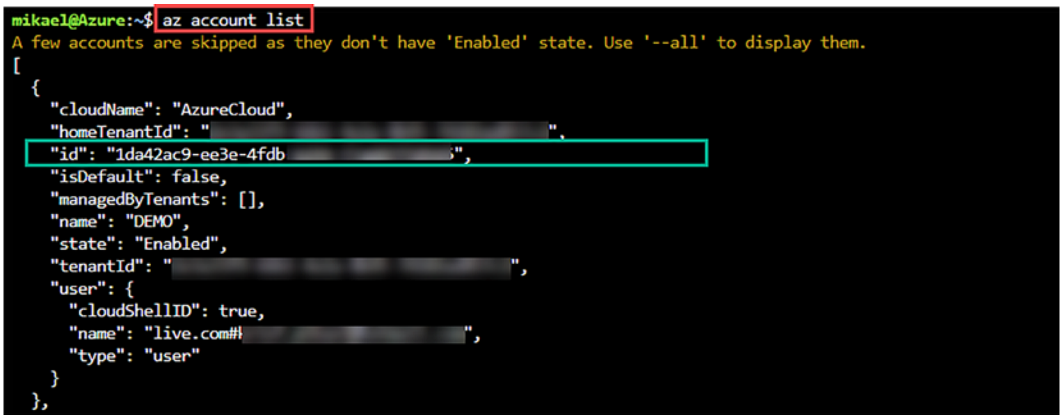
Therefore, one of the options we have when running Terraform in Azure is to use an app registration account (also known as a service principal) that is not linked to a physical person.

In this recipe, we will first study the creation of this service principal and then we will see how to use it securely to run a Terraform configuration.

Getting ready

To apply the first part of this recipe, you must have user account creation permissions in Azure Active Directory. Moreover, to create this `service principal`, we will do it using the command line with the Azure CLI tool, documentation relating to which is available at `https://docs.microsoft.com/cli/azure/?view=azure-cli-latest`.

In addition, we need to retrieve the ID of the subscription in which resources will be provisioned. For this, we can get it in Azure Cloud Shell by running the `az account list` command to display our subscription details:



```
mikael@Azure:~$ az account list
A few accounts are skipped as they don't have 'Enabled' state. Use '--all' to display them.
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "[redacted]",
    "id": "1da42ac9-ee3e-4fdb[redacted]",
    "isDefault": false,
    "managedByTenants": [],
    "name": "DEMO",
    "state": "Enabled",
    "tenantId": "[redacted]",
    "user": {
      "cloudShellID": true,
      "name": "live.com#[redacted]",
      "type": "user"
    }
  },
```

Figure 8.4: Getting the account ID using Azure CLI

We can then get the `id` property of the subscription concerned by performing the copy/paste operation.

You can also run the command `az account show --query id --output tsv` to get only the `Subscription ID` value.

How to do it...

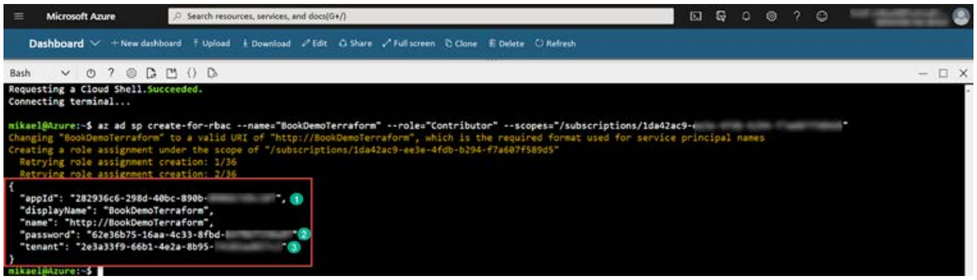
This recipe comprises two parts: part one is the creation of the service principal, and part two is the configuration of Terraform authentication using this service principal.

To create this service principal, perform the following steps:

1. Open Azure Cloud Shell and execute the following command:

   ```
   az ad sp create-for-rbac --name="BookDemoTerraform"
   --role="Contributor" --scopes="/subscriptions/<Subscription Id>"
   ```

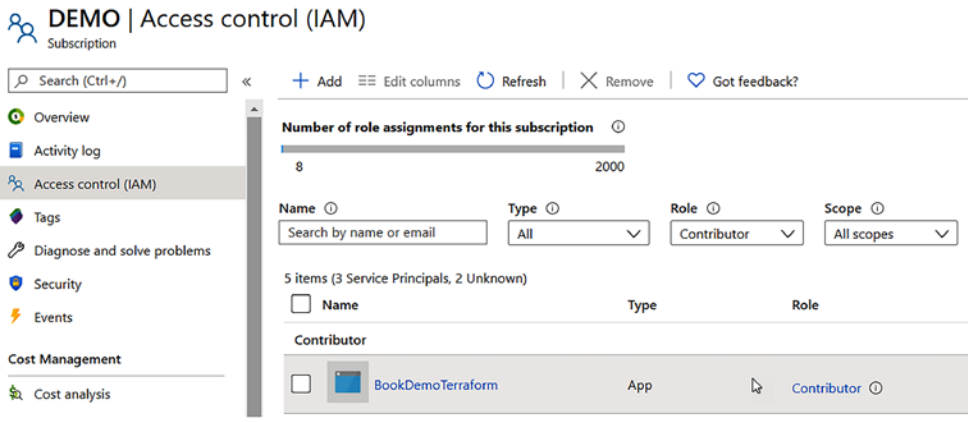
2. Retrieve all the identification information provided in the output of the previous command by making a note (because we won't be able to retrieve the password after closing this console) of `appId`, `password`, and `tenant`, as shown in the following screenshot:



```
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

mikael@Azure:~$ az ad sp create-for-rbac --name="BookDemoTerraform" --role="Contributor" --scopes="/subscriptions/1da42ac9-[redacted]"
Changing "BookDemoTerraform" to a valid URI of "http://BookDemoTerraform", which is the required format used for service principal names
Creating a role assignment under the scope of "/subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5"
  Retrying role assignment creation: 1/36
  Retrying role assignment creation: 2/36
{
  "appId": "282936c6-298d-40bc-890b-[redacted]", 1
  "displayName": "BookDemoTerraform",
  "name": "http://BookDemoTerraform",
  "password": "62e36b75-16aa-4c33-8fbd-[redacted]" 2
  "tenant": "2e3a33f9-66b1-4e2a-8b95-[redacted]" 3
}
mikael@Azure:~$
```

Figure 8.5: Getting account information using the Azure CLI

3. We check that the service principal has the "contributor" permissions on the subscription: to perform this, in the Azure portal, click on the **Access control (IAM)** link on the left menu on the **Subscription** details page, as you can see in the following screenshot:



Number of role assignments for this subscription

8 2000

Name: Search by name or email
Type: All
Role: Contributor
Scope: All scopes

5 items (3 Service Principals, 2 Unknown)

Name	Type	Role
Contributor		
BookDemoTerraform	App	Contributor

Figure 8.6: Showing the service principal in Access control (IAM)

We can see in the preceding image that the service principal `BookDemoTerraform` is a contributor to the subscription.

Now that the service principal is created and has contributor permissions on the subscription, we can use it to provision Azure infrastructure with Terraform by performing the following steps:

1. In the command-line terminal, set four new environments variables as follows:

```
export ARM_SUBSCRIPTION_ID=<subscription_id>
export ARM_CLIENT_ID=<appId>
export ARM_CLIENT_SECRET=<password>
export ARM_TENANT_ID=<tenant id>
```

2. Then, we can apply the Terraform configuration by executing the following Terraform workflow:

```
terraform init
terraform plan -out=app.tfplan
terraform apply app.tfplan
```

How it works...

In the first part of this recipe, we created a service principal and gave it its permissions on a subscription, using the command `az ad sp`. We added the following arguments to this command:

- `name`, which is the name of the service principal we are going to create.
- `role`, which is the role that the service principal will have on the subscription; here, we specify `Contributor`.
- `scopes`, where we specify the Azure ID of the resource on which the service principal will have contributor permissions. In our case, this is the subscription ID in which the resources will be provisioned by Terraform.

This command will therefore create the service principal with a generated password and will give it the `Contributor` role on the specified subscription.

At the end of its execution, this command displays the information of the service principal, including `AppId`, `password`, and `tenant`. As explained in *Step 2*, we need to retrieve this information and store it in a safe place because this password cannot be retrieved later. Then, we check via the Azure portal that the service principal has the contributor permissions on the subscription.

In the second part of this recipe, we used this service principal to authenticate the Terraform Azure provider. For this, there are several solutions, the most secure one being to use specific Azure provider environment variables because these environment variables will not be visible in the code and will only be persistent during the execution session. So, we have set four environment variables, which are as follows:

- `ARM_SUBSCRIPTION_ID`: This contains the Azure subscription ID.
- `ARM_CLIENT_ID`: This contains the service principal ID, called `AppId`.
- `ARM_CLIENT_SECRET`: This contains the password of the service principal.
- `ARM_TENANT_ID`: This contains the ID of the Azure Active Directory tenant.

In the recipe, we used the `export` command of the Linux system. On Windows PowerShell, we can use the `$env` command. In addition, in all subsequent recipes in this chapter, we will use these environment variables before executing Terraform.

Once these environment variables were set, we executed the basic Terraform workflow with `terraform init`, `plan`, and `apply`.

There's more...

Regarding the creation of the service principal, we made the choice to use the Azure CLI tool, but we could also have done it directly via the Azure portal, as detailed in the documentation available at `https://docs.microsoft.com/azure/active-directory/develop/howto-create-service-principal-portal`, or we could have used the Azure PowerShell commands (`https://docs.microsoft.com/azure/active-directory/develop/howto-authenticate-service-principal-powershell`).

In addition, as regards the configuration of the Terraform Azure provider, we have used environment variables, but we can also put this information directly into the Terraform configuration, as shown in the following code snippet:

```
provider "azurerm" {
   ...
  subscription_id = "<Subscription ID>"
  client_id       = "<Client ID>"
  client_secret   = "<Client Secret>"
  tenant_id       = "<Tenant ID>"
}
```

This solution does not require an extra step (the set of environment variables) to be implemented prior to executing the Terraform configuration, but it leaves identification information in clear text in the code, and hardcoding credentials in code is generally considered a bad practice from a security perspective since the leakage of code also leaks credentials and makes it impossible to share the code with anyone without exposing the credentials. And in the case where the Terraform configuration provides resources for several environments that are in different subscriptions, Terraform variables will have to be added, which can add complexity to the code.

Finally, the use of environment variables offers the advantage of being easily integrated into a CI/CD pipeline while preserving the security of the authentication data.

See also

- The Azure documentation and Terraform configuration are available here: `https://docs.microsoft.com/azure/developer/terraform/install-configure`
- Documentation on the configuration of the `azurerm` provider, along with the other authentication options, is available here: `https://www.terraform.io/docs/providers/azurerm/index.html`

Protecting the state file in the Azure remote backend

When executing the Terraform workflow commands, which are mainly `terraform plan`, `terraform apply`, and `terraform destroy`, Terraform has a mechanism that allows it to identify which resources need to be updated, added, or deleted. To perform this mechanism, Terraform maintains a file called a Terraform state file that contains all the details of the resources provisioned by Terraform. This Terraform state file is created the first time the `terraform plan` command is run and is updated with each action (`apply` or `destroy`).

For more information about how to manage the Terraform state, read *Chapter 5*, *Managing Terraform State*.

In an enterprise, the Terraform state file can present certain interesting problems:

- Sensitive information on the provisioned resources is mentioned in clear text.
- If several people are working together, this file must be shared by everyone, or, by default, this file is created on the local workstation or on the workstation that contains the Terraform binary.
- Even if it is archived in a Git repository, once it is retrieved on the local workstation, it does not allow several people to work on the same file.
- With this locally stored file, managing multiple environments can quickly become complicated and risky.
- Any deletion of this local file or manual editing can affect the execution of the Terraform configuration.

A solution to all these problems is the use of a remote backend, which consists of storing this file in a remote, shared, and secure store.

In the case of Terraform, there are several types of remote backends, such as S3, `azurerm`, Artifactory, and many others, which are listed in the menu on the following page: `https://www.terraform.io/docs/backends/types/index.html`.

In this recipe, we will study the use of a remote backend in Azure, `azurerm`, by storing this Terraform state file inside a container in an Azure Storage Account.

Getting ready

For this recipe, we will use Azure Cloud Shell and Azure CLI commands to create the Storage Account.

The source code for this recipe and the script used are available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/remotebackend`.

How to do it…

This recipe consists of three parts. In the first part, we will create the Storage Account, in the second part, we will configure Terraform to use the `azurerm` remote backend, and finally, we will set the `ARM_ACCESS_KEY` environment variable:

1. In Azure Cloud Shell, execute the following Azure CLI script (available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP08/remotebackend/create-backend.sh`) to create the Storage Account with a blob container in the resource group:

```
# 1- Create Resource Group
az group create --name "RG-TFBACKEND" --location westeurope

# 2- Create storage account
az storage account create --resource-group "RG-TFBACKEND" --name
"storagetfbackend " --sku Standard_LRS --encryption-services blob

# 3- Create blob container
az storage container create --name "tfstate" --account-name
"storagetfbackend"

# 4- Get storage account key
ACCOUNT_KEY=$(az storage account keys list --resource-group "RG-
TFBACKEND" --account-name "storagetfbackend" --query [0].value -o
tsv)
```

```
echo $ACCOUNT_KEY
```

Note that in the preceding script, the name of the storage account, `storagetfbackend`, is just an example. When you execute this script, change the name to a unique storage name.

2. Then, we configure the Terraform state backend by adding the following code to the `main.tf` file:

```
terraform {
  backend "azurerm" {
    resource_group_name  = "RG-TFBACKEND"
    storage_account_name = "storagetfbackend" # name of your Azure
Storage Account
    container_name       = "tfstate"
    key                  = "myapp.tfstate"
  }
}
```

3. Finally, to execute the Terraform configuration, we set a new environment variable, `ARM_ACCESS_KEY`, with the following command:

```
export ARM_ACCESS_KEY=<access key>
```

4. Set the four authentication environment variables we learned about in the *Protecting the Azure credential provider* recipe in this chapter.
5. Finally, we execute the basic commands of the Terraform workflow.

How it works...

In the first step, we used a script that performs the following actions in sequence:

1. It creates a resource group called `RG-TFBACKEND`.
2. In this resource group, we use the `az storage account create` command to create a Storage Account named `storagetfbackend`.
3. Then, this script creates a blob container in this Storage Account with the `az storage container create` command.
4. Finally, we retrieve the account key of the Storage Account created and display its value.

Then, in *Step 2*, we configure Terraform to use this Storage Account as a remote backend to store the Terraform state file. In this configuration, which is located in a `backend "azurerm"` block, we indicate, on the one hand, the Storage Account information, and on the other, the blob with the following properties:

- `resource_group_name`: This is the name of the resource group that contains the Storage Account.
- `storage_account_name`: This is the name of the Storage Account.
- `container_name`: This is the name of the blob container.
- `key`: This is the name of the Terraform state file.

Finally, we define a new environment variable, `ARM_ACCESS_KEY`, that contains the account key for the Storage Account we retrieved from the script we ran in step 1. This variable is used to authenticate the Storage Account.

Then, to set all the environment variables, we can execute the `init`, `plan`, and `apply` commands of Terraform.

Based on what we studied in the previous recipe, *Protecting the Azure credential provider*, here is the complete script for executing this Terraform script in Azure:

```
export ARM_SUBSCIPTION_ID =<subscription_id>
export ARM_CLIENT_ID=<appId>
export ARM_CLIENT_SECRET=<password>
export ARM_TENANT_ID=<tenant id>
export ARM_ACCESS_KEY=<account key>

terraform init
terraform plan -out=app.tfplan
terraform apply app.tfplan
```

So, we used the four authentication environment variables, as well as the `ARM_ACCESS_KEY` environment variable, for authentication to the Storage Account, and we executed the Terraform commands.

There's more...

In this recipe, we used an environment variable to specify the value of the access key to protect this sensitive data.

We could have specified it in the remote backend configuration using the `access_key` property, as in the following example, but as mentioned in the *Protecting the Azure credential provider* recipe of this chapter, it isn't good practice to leave sensitive keys as plain text:

```
terraform {
  backend "azurerm" {
    resource_group_name  = "RG-TFBACKEND"
    storage_account_name = "storagetfbackend"
    container_name       = "tfstate"
    key                  = "myapp.tfstate"
    access_key           = xxxxxx-xxxxx-xxx-xxxxx
  }
}
```

Moreover, if our Terraform configuration is designed to be deployed on multiple environments, we can create N configurations of the `azurerm` backend with the following steps:

1. The `main.tf` file contains the following code with the `backend "azurerm"` block empty:

   ```
   terraform {
     required_version = ">= 1.0"
     backend "azurerm" {
     }
   }
   ```

2. We create one `backend.tfbackend` file per environment (in a specific folder for this environment) with the following code:

   ```
   resource_group_name  = "RG-TFBACKEND"
   storage_account_name = "storagetfbackend"
   container_name       = "tfstate"
   key                  = "myapp.tfstate"
   ```

3. Finally, in the execution of the `init` command, we specify the `backend.tfbackend` file to be used with the following command, as specified in the `init` command documentation, which is available at `https://www.terraform.io/docs/backends/config.html#partial-configuration`:

   ```
   terraform init -backend-config="<path>/backend.tfbackend"
   ```

Another consideration is that if the service principal that was used to authenticate with Terraform has permissions on this Storage Account, then this environment variable is not mandatory.

See also

- Documentation relating to the `azurerm` remote backend is available here: `https://developer.hashicorp.com/terraform/language/settings/backends/azurerm`
- Terraform's learning module with the `azurerm` remote backend is available here: `https://learn.hashicorp.com/terraform/azure/remote_az#azurerm`
- Azure documentation relating to the Terraform remote backend is available here: `https://docs.microsoft.com/azure/developer/terraform/store-state-in-azure-storage`

Executing ARM templates in Terraform

Among all the **Infrastructure as Code** (**IaC**) tools and languages, there is one provided by Azure called ARM, based on JSON format files, that contains the description of the resources to be provisioned.

To learn more about ARM templates, read the following documentation: `https://docs.microsoft.com/azure/azure-resource-manager/templates/overview`.

When using Terraform to provision resources in Azure, you may need to use resources that are not yet available in the Terraform `azurerm` provider. Indeed, the `azurerm` provider is open source and community-based on GitHub and has a large community of contributors, but this is not enough to keep up with all the changes in Azure's functionalities in real time. This is due to several reasons:

- New releases of Azure resources are very frequent.
- The Terraform `azurerm` provider is highly dependent on the Azure Go SDK (`https://github.com/Azure/azure-sdk-for-go`), which does not contain real-time Azure updates on new features or even on features that are still in preview.

To partially solve this problem, and for organizations that wish to remain in full Terraform, there is a Terraform `azurerm_template_deployment` resource that allows you to execute ARM code using Terraform.

In this recipe, we will discuss the execution of ARM code with Terraform.

Getting ready

The Terraform configuration of this recipe will provision an Azure App Service resource that includes an extension. Since the extension App Service resource is not available in the `azurerm` provider at the time of writing this book, the Azure App Service code will be written in **HashiCorp Configuration Language** (**HCL**), and its extension will be provisioned using an ARM template that will be executed with Terraform.

The purpose of the recipe is not to detail the code of the ARM template of the extension but to study its execution with Terraform.

In this recipe, we will present only the key code extracts. The complete source code for this chapter is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/arm-template`.

How to do it…

To execute the ARM template with Terraform, perform the following steps:

1. Inside the folder that will contain the Terraform configuration, create a new file called `ARM_siteExtension.json`, which contains the following ARM JSON template:

```
{
...
  "parameters": {
    "appserviceName": { ... },
    "extensionName": { ... },
    "extensionVersion": { ... }
  },
  "resources": [
    {
      "type": "Microsoft.Web/sites/siteextensions",
      "name": "[concat(parameters('appserviceName'), '/',
parameters('extensionName'))]",
      ...
      "properties": {
        "version": "[parameters('extensionVersion')]"
      }
    }
  ]
}
```

The complete source code of this file is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP08/arm-template/ARM_siteExtension.json`.

2. In the `main.tf` file, add the following Terraform extract code:

```
resource "azurerm_resource_group_template_deployment" "extension" {
  name                = "extension"
  resource_group_name = azurerm_resource_group.rg-app.name
  template_content    = file("ARM_siteExtension.json")

  parameters_content = jsonencode({
    "appserviceName" = {
      value = azurerm_linux_web_app.app.name
    },
    "extensionName" = {
      value = "AspNetCoreRuntime.2.2.x64"
    },
    "extensionVersion" = {
      value = "2.2.0-preview3-35497"
    }
  })

  deployment_mode = "Incremental"
}
```

The complete source code of this file is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP08/arm-template/main.tf`.

3. Then, we can execute the basic Terraform workflow with the following:

    - Authentication with four Azure environment variables, as discussed in the *Protecting the Azure credential provider* recipe of this chapter
    - The execution of the `init`, `plan`, and `apply` commands, as mentioned previously and in earlier chapters

How it works...

In *Step 1*, in the directory that contains the Terraform configuration, we created a JSON file that contains the ARM code for creating an extension for an App Service resource. In this ARM file, we have the following three input parameters:

- `appserviceName`: This corresponds to the name of the App Service resource.
- `extensionName`: This corresponds to the name of the extension to be added (from the extension catalog).
- `extensionVersion`: This corresponds to the version of the extension to be added.

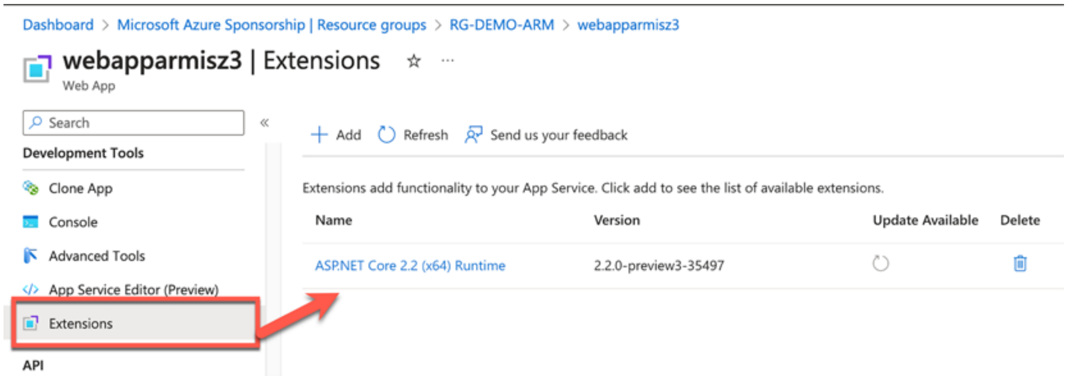
Then, the rest of this file describes the site extension resource to be added in Azure using the three parameters.

Then, in *Step 2*, in the Terraform configuration, we used the Terraform resource `azurerm_resource_group_template_deployment`, which allows the execution of an ARM template with the following properties:

- `template_content`: This is the ARM code in JSON format. Here, in our example, we used the `file` function to indicate that it is a file.
- `parameters_content`: In this block, we fill in the input properties of the ARM template in JSON format, which are `appserviceName`, `extensionName`, and `extensionVersion`. In our recipe, we install the `AspNetCoreRuntime.2.2.x64` extension of version `2.2.0-preview3-35497`.

Finally, to provision this Azure App Service resource and its extension, the Terraform workflow commands are executed.

The following screenshot shows the result of executing the workflow commands in the Azure portal:


Dashboard > Microsoft Azure Sponsorship | Resource groups > RG-DEMO-ARM > webapparmisz3
webapparmisz3 | Extensions
Web App
Search
Development Tools
Clone App
Console
Advanced Tools
App Service Editor (Preview)
Extensions
API
Add
Refresh
Send us your feedback

Extensions add functionality to your App Service. Click add to see the list of available extensions.

Name	Version	Update Available	Delete
ASP.NET Core 2.2 (x64) Runtime	2.2.0-preview3-35497		

Figure 8.7: Azure App Service extension

We can see the extension provisioned inside the App Service resource.

There's more...

In this recipe, we have studied the possibility of running an ARM template with Terraform. This method allows you to provision elements in Azure that are not available in the `azurerm` provider, but it is important to know that Terraform knows the resources described in this ARM template when it is executed.

That is to say that these resources (here, in our Resource, it is the extension) do not follow the lifecycle of the Terraform workflow and are not registered in the Terraform state file. For this reason, it is advisable that you use this type of deployment only to complete resources that have been provisioned with Terraform HCL code.

In the next recipe, we will stay on the same topic of handling a provisioned resource that isn't available in the `azurerm` provider yet, but instead of using an ARM template, we will see how to use Azure CLI commands with Terraform.

See also

- Documentation pertaining to the `azurerm_resource_group_template_deployment` resource of the `azurerm` provider is available here: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/resource_group_template_deployment`

Executing Azure CLI commands in Terraform

In the previous recipe, we studied how to run ARM templates with Terraform in a situation where the provisioned resource is not yet available in the `azurerm` provider.

However, there are cases where the use of an ARM template is not possible, such as the following:

- We want to fill in one or more properties of a resource, which are not available in an ARM template.
- The ARM template is not available for the resource to be provisioned.

For these situations, there is another solution, which entails executing Azure CLI commands with Terraform.

This recipe is a practical application of the *Executing local programs with Terraform recipe* from *Chapter 4*, *Using Terraform with External Data*. We will study the Terraform configuration and its execution to integrate Azure CLI commands with Terraform.

Getting ready

For this recipe, it is necessary to have read beforehand the *Executing local programs with Terraform recipe* from *Chapter 4*, *Using Terraform with External Data*, which provides the basis of the Terraform configuration we are going to write.

Moreover, it will also be necessary to have already installed the Azure CLI tool, documentation pertaining to which is available here: `https://docs.microsoft.com/cli/azure/?view=azure-cli-latest`.

To demonstrate the use of the Azure CLI command in Terraform, in this recipe, we will set up an Azure Storage Account by configuring the properties of a static website feature in it.

As with the previous recipe, the purpose of this recipe is to show how to use Azure CLI commands with Terraform, but we will not focus on the Azure CLI command used because, since version 2.0.0 of the `azurerm` provider, the properties of a static website have been added to the Terraform resource (`https://github.com/terraform-providers/terraform-provider-azurerm/blob/master/CHANGELOG-v2.md#200-february-24-2020`).

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/azcli`.

How to do it...

Perform the following steps to execute Azure CLI commands with Terraform:

1. In the `main.tf` file that contains the Terraform configuration, write the following configuration to provision the Storage Account:

   ```
   resource "azurerm_storage_account" "sa" {
     name                     = "saazclidemo"
     resource_group_name      = azurerm_resource_group.rg.name
     location                 = "westeurope"
     account_tier             = "Standard"
     account_kind             = "StorageV2"
     account_replication_type = "GRS"
   }
   ```

2. In the same Terraform configuration, add the code to configure the static website using the following Azure CLI command:

   ```
   resource "null_resource" "webapp_static_website" {
     triggers = {
       account = azurerm_storage_account.sa.name
     }

     provisioner "local-exec" {
       command = "az storage blob service-properties update --account-
   name ${azurerm_storage_account.sa.name} --static-website true
   --index-document index.html --404-document 404.html"
     }
   }
   ```

3. Then, in our command-line terminal, we log in to Azure by executing the following command by replacing your `APP_ID` (`CLIENT ID`), `PASSWORD` (`CLIENT SECRET`) and `TENANT_ID`:

   ```
   az login --service-principal --username APP_ID --password PASSWORD
   --tenant TENANT_ID
   ```

4. Finally, we can execute the basic Terraform workflow with the following:

   - Authentication with four Azure environment variables, as discussed in the *Protecting the Azure credential provider* recipe of this chapter

- The execution of the `init`, `plan`, and `apply` commands, as mentioned previously and in earlier chapters

How it works...

In *Step 1*, there is nothing special. We just wrote the Terraform configuration to provision a `StorageV2` Storage Account, which is required to activate the static website feature.

In *Step 2*, we completed this code by adding `null_resource`, which contains a `local-exec` provisioner. In the command property of `local-exec`, we run the command Azure CLI that must be executed to activate and configure the static website functionality on the Storage Account we wrote in *Step 1*.

We added the trigger block with the name of the storage as an argument, so that if the name of the storage changes, then provisioning will be re-executed.

Then, in *Step 3*, we executed the `az login` command to authenticate the context of the Azure CLI. On this command, we added the authentication parameters with a service principal (see the Protecting the Azure credential provider recipe in this chapter), as documented here: `https://learn.microsoft.com/en-us/cli/azure/authenticate-azure-cli#sign-in-with-a-service-principal`.

There's more...

In the implementation of this recipe, there are two important points:

- The first point is that we used `null_resource` with the `local-exec` provisioner that we had already studied in detail in the *Executing local programs with Terraform recipe* from *Chapter 4*, *Using Terraform with External Data*. The only novelty brought here is the fact that the executed command is an Azure CLI command. It could also be a file that contains a script with several Azure CLI commands.

The second point is that Terraform's authentication for Azure with the four environment variables does not allow authentication of the Azure CLI context that will be executed by Terraform. This is why, in *Step 3*, we also had to authenticate the Azure CLI context with the `az login` command by passing the credentials of the service principal as parameters.

The advantage of executing Azure CLI commands in this way is that we can integrate the variables and expressions of the Terraform language into them, just as we did when we passed the name of the Storage Account as a parameter.

Note that, as with any local provisioner, this restricts where the configuration can be applied as it assumes the existence of the Azure CLI (the Azure CLI becomes a hidden dependency).

As with the ARM templates we learned about in the previous recipe, *Executing ARM templates in Terraform*, Terraform does not know the resources manipulated in the Azure CLI command or script. These resources do not follow the Terraform lifecycle and are not registered in the Terraform state file. On the other hand, in the `local-exec` provisioner of `null_resource`, we can specify a command to be executed in the case of execution of the `terraform destroy` command.

The following is an example of the configuration that I used to create a Cosmos DB database (before the `azurerm` provider supported it):

```
resource "null_resource" "cosmosdb_database" {
  provisioner "local-exec" {
    command = "az cosmosdb database create --name ${var.cosmosdb_
name} --db-name ${var.app_name}   --resource-group ${var.cosmosdb_rg}
--throughput ${var.cosmosdb_throughput}"
  }

  provisioner "local-exec" {
    when    = "destroy"
    command = "az cosmosdb database delete --name ${var.cosmosdb_name}
--db-name ${var.app_name}   --resource-group ${var.cosmosdb_rg}"
  }
}
```

In this example, in the provisioner, we used the `when = "destroy"` property to specify that the Azure CLI command, `az cosmosdb database delete`, will be executed to delete the Cosmos DB database in the case of `terraform destroy`.

See also

- Documentation pertaining to the `az login` command and its parameters is available here: `https://docs.microsoft.com/cli/azure/authenticate-azure-cli?view=azure-cli-latest`
- Documentation pertaining to the Terraform `provisioner` is available here: `https://www.terraform.io/docs/provisioners/index.html`

- Documentation pertaining to the `when` property of `provisioner` is available here: `https://www.terraform.io/docs/provisioners/index.html#destroy-time-provisioners`

Using Azure Key Vault with Terraform to protect secrets

One of the challenges of IaC is the protection of sensitive information that is part of the infrastructure.

Indeed, one of the advantages of IaC is the possibility to version the code in a Git repository and so this code benefits from the Git workflow of versioning and validation of the code. However, with this approach, we tend to write everything in this code, sometimes forgetting that some data that is sensitive, such as passwords or login strings, can be misused if they end up in the wrong hands.

In this recipe, we will study how to protect this sensitive data by storing it in Azure's secret manager, which is Azure Key Vault, and then using it in the Terraform configuration.

Getting ready

For this recipe, we assume the use of Azure Key Vault. For more information, you can refer to the documentation available at `https://docs.microsoft.com/azure/key-vault/`.

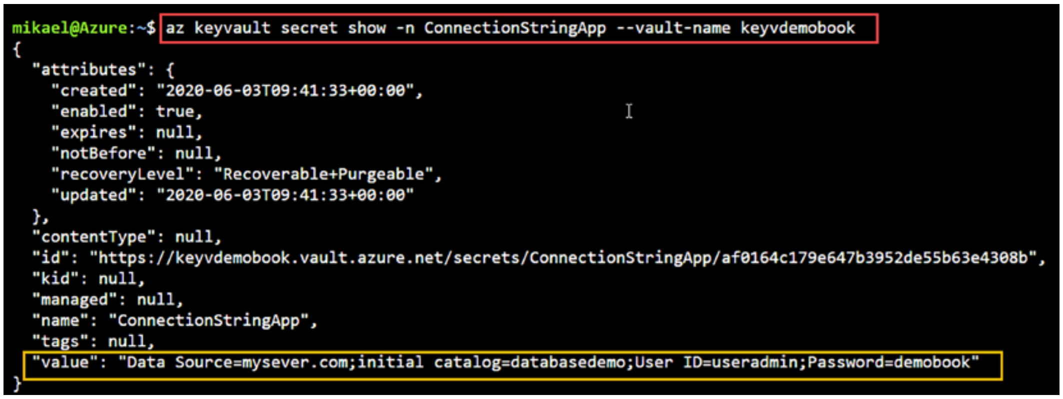
As a prerequisite for this recipe, we need to manually create the secret of the connection string in Azure Key Vault. This can be done either with Terraform, as documented here (`https://www.terraform.io/docs/providers/azurerm/r/key_vault_secret.html`), or with the Azure CLI commands, as documented here (`https://learn.microsoft.com/en-gb/cli/azure/keyvault/secret?view=azure-cli-latest#az-keyvault-secret-set`).

In the Azure Key Vault instance that we have created for the application of this recipe, we store a secret that protects the connection string of the SQL Server database of our application hosted in an Azure web application.

This connection string is as follows:

```
Data Source=mysever.com;initial catalog=databasedemo;User
ID=useradmin;Password=demobook
```

Here is the output of the Azure CLI command, `az keyvault secret show`, which shows its storage and properties in Azure Key Vault:



```
mikael@Azure:~$ az keyvault secret show -n ConnectionStringApp --vault-name keyvdemobook
{
  "attributes": {
    "created": "2020-06-03T09:41:33+00:00",
    "enabled": true,
    "expires": null,
    "notBefore": null,
    "recoveryLevel": "Recoverable+Purgeable",
    "updated": "2020-06-03T09:41:33+00:00"
  },
  "contentType": null,
  "id": "https://keyvdemobook.vault.azure.net/secrets/ConnectionStringApp/af0164c179e647b3952de55b63e4308b",
  "kid": null,
  "managed": null,
  "name": "ConnectionStringApp",
  "tags": null,
  "value": "Data Source=mysever.com;initial catalog=databasedemo;User ID=useradmin;Password=demobook"
}
```

Figure 8.8: Azure Key Vault secret

In the preceding screenshot, we can see the connection string of the database stored in the `value` property of the `secret` object.

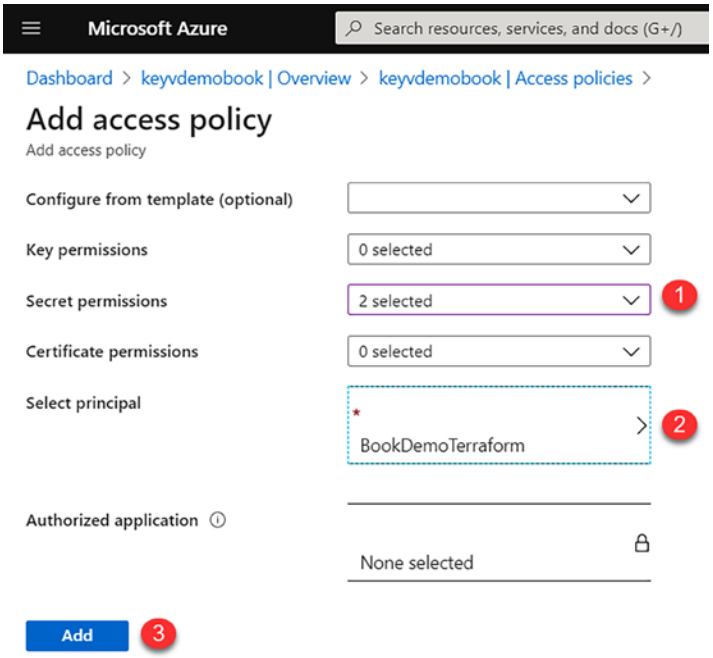
The goal of this recipe is to write the Terraform configuration that requests the value of this secret to use it in the properties of an Azure App Service instance.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/keyvault`.

How to do it...

To get and use an Azure Key Vault secret in Terraform, perform the following steps:

1. In Azure Key Vault, we add access policy properties by granting the service principal that will be used by Terraform for Azure (for more details about the creation of the service principal for Terraform, read the *Protecting the Azure credential provider* recipe in this chapter) to have permission to get and list secrets:


Microsoft Azure
Search resources, services, and docs (G+/)
Dashboard > keyvdemobook | Overview > keyvdemobook | Access policies >
1
2
3

Add access policy

Add access policy

Configure from template (optional)

Key permissions: 0 selected

Secret permissions: 2 selected

Certificate permissions: 0 selected

Select principal: * BookDemoTerraform

Authorized application: None selected

Add

Figure 8.9: Add access policy

Note that to make it more visual, in this recipe, we set the access policy of the Azure Key Vault using the Azure Portal, but it is more recommended to do it in the Terraform configuration by referring to this documentation: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/key_vault_access_policy`.

2. In the `main.tf` file, we add the following code to get the Key Vault secret:

```
data "azurerm_key_vault" "keyvault" {
  name                = "keyvdemobook"
  resource_group_name = "rg_keyvault"
}

data "azurerm_key_vault_secret" "app-connectionstring" {
  name         = "ConnectionStringApp"
  key_vault_id = data.azurerm_key_vault.keyvault.id
}
```

3. Then, in the Terraform configuration of the App Service resource, in the `main.tf` file, we add the following code:

```
resource "azurerm_linux_web_app" "app" {
  name                = "demovaultbook-${random_string.random.
result}"
  location            = azurerm_resource_group.rg-app.location
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id     = azurerm_service_plan.plan-app.id

  connection_string {
    name  = "Database"
    type  = "SQLServer"
    value = data.azurerm_key_vault_secret.app-connectionstring.value
  }

  site_config {}
}
```

4. Finally, we run the basic Terraform workflow for Azure with the four environment variables (from the *Protecting the Azure credential provider* recipe of this chapter) and execute `init`, `plan`, and `apply`, as mentioned previously and in earlier chapters.

How it works...

In *Step 1*, we gave permission to the service principal used by Terraform to read and list the secrets of Azure Key Vault.

> We can do this either via the Azure portal or on the command line with the Azure CLI, as explained in the following documentation: `https://docs.microsoft.com/cli/azure/keyvault?view=azure-cli-latest#az-keyvault-set-policy`.

Then, in *Step 2*, we wrote the Terraform configuration, which contains two data sources:

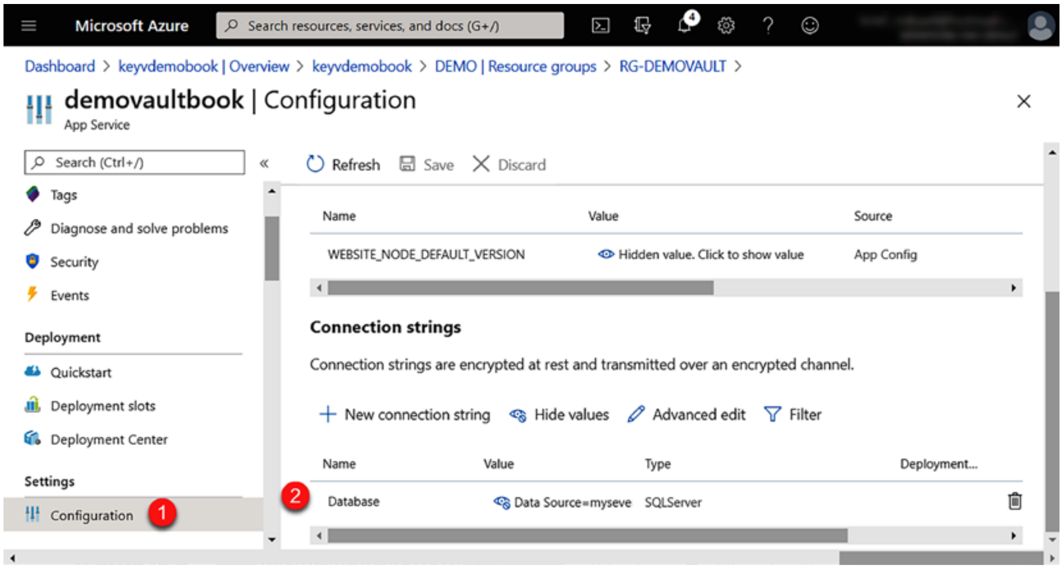
- The first data source, `azurerm_key_vault`, enables retrieval of the Azure ID of the Azure Key Vault resource.
- The second data source, `azurerm_key_vault_secret`, is used to retrieve the secret that contains the database connection string as a value.

For more information about Terraform block data, read the *Using external resources with a data block* recipe from *Chapter 4*, *Using Terraform with External Data*.

In *Step 3*, we continue with the writing of the Terraform configuration, putting in the property value of the connection_string block of the App Service resource with the expression data.azurerm_key_vault_secret.app-connectionstring.value, which is the value obtained from the block data, azurerm_key_vault_secret, written in *Step 2*.

Finally, in the last step, we execute this Terraform configuration. During this operation, Terraform will first retrieve the values requested in the block data (Key Vault, and then the Key Vault secret) and will then inject the value obtained from the secret into the configuration of the App Service resource.

This result is obtained in Azure and shown in the following screenshot:


Microsoft Azure
Search resources, services, and docs (G+/)
Dashboard > keyvdemobook | Overview > keyvdemobook > DEMO | Resource groups > RG-DEMOVAULT >
demovaultbook | Configuration
App Service
Search (Ctrl+/)
Tags
Diagnose and solve problems
Security
Events
Deployment
Quickstart
Deployment slots
Deployment Center
Settings
Configuration
1
Refresh
Save
Discard
2

Name	Value	Source
WEBSITE_NODE_DEFAULT_VERSION	Hidden value. Click to show value	App Config

Connection strings

Connection strings are encrypted at rest and transmitted over an encrypted channel.

Name	Value	Type	Deployment...
Database	Data Source=myseve	SQLServer	

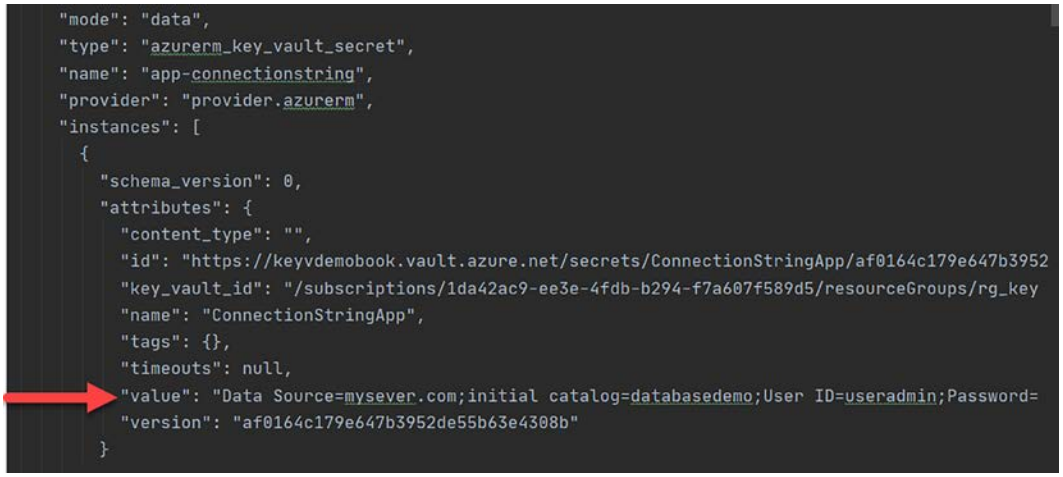
Figure 8.10: Azure App Service configuration

We can see that the connection string is well filled in the App Service configuration.

There's more...

We have learned in this recipe that the connection string, which contains sensitive data, has been stored in Azure Key Vault and will be used automatically when Terraform is run. So, thanks to Azure Key Vault, we didn't need to put the sensitive data in clear text in the Terraform configuration.

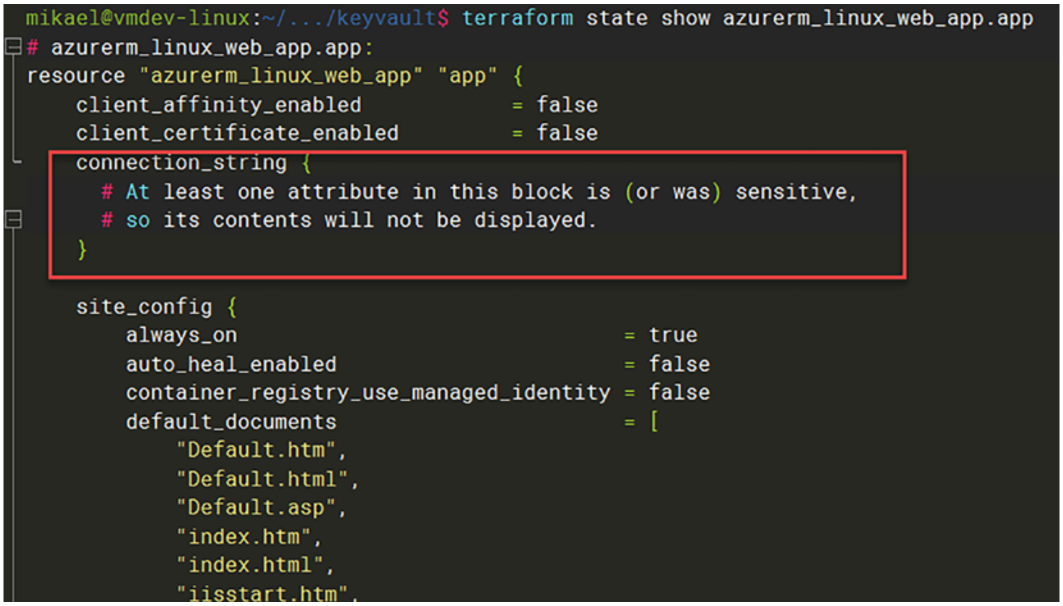
However, care should still be taken. Although this data is not written in plain text in the Terraform configuration, it will be written in plain text in the Terraform state file, as can be seen in this extract of the Terraform state file content from this recipe:



```
"mode": "data",
"type": "azurerm_key_vault_secret",
"name": "app-connectionstring",
"provider": "provider.azurerm",
"instances": [
  {
    "schema_version": 0,
    "attributes": {
      "content_type": "",
      "id": "https://keyvdemobook.vault.azure.net/secrets/ConnectionStringApp/af0164c179e647b3952
      "key_vault_id": "/subscriptions/1da42ac9-ee3e-4fdb-b294-f7a607f589d5/resourceGroups/rg_key
      "name": "ConnectionStringApp",
      "tags": {},
      "timeouts": null,
      "value": "Data Source=mysever.com;initial catalog=databasedemo;User ID=useradmin;Password=
      "version": "af0164c179e647b3952de55b63e4308b"
    }
```

Figure 8.11: Sensitive data in Terraform state

That is why, if we need to inspect the contents of this file, it is recommended to use the `terraform state show  azurerm_linux_web_app.app` command, which protects the displaying of sensitive data, as can be seen in the following screenshot:



```
mikael@vmdev-linux:~/.../keyvault$ terraform state show azurerm_linux_web_app.app
# azurerm_linux_web_app.app:
resource "azurerm_linux_web_app" "app" {
    client_affinity_enabled           = false
    client_certificate_enabled        = false
    connection_string {
      # At least one attribute in this block is (or was) sensitive,
      # so its contents will not be displayed.
    }

    site_config {
        always_on                               = true
        auto_heal_enabled                       = false
        container_registry_use_managed_identity = false
        default_documents                       = [
            "Default.htm",
            "Default.html",
            "Default.asp",
            "index.htm",
            "index.html",
            "iisstart.htm",
```

Figure 8.12: Terraform protecting sensitive data

This is one of the reasons why it is necessary to protect this Terraform state file by storing it in a secure remote backend, as we have seen in the *Protecting the state file in the Azure remote backend* recipe of this chapter, and which is explained in the following documentation: `https://www.terraform.io/docs/state/sensitive-data.html`.

Also in this recipe, although we stored the sensitive data in Azure Key Vault, we can also store it in a HashiCorp Vault instance that integrates very well with Terraform. For this, it is advised that you read the Vault provider documentation here: `https://www.terraform.io/docs/providers/vault/index.html`.

Finally, as a prerequisite for this recipe, we manually created the secret of the connection string in Azure Key Vault. This could have been done either with Terraform, as documented here (`https://www.terraform.io/docs/providers/azurerm/r/key_vault_secret.html`), or with the Azure CLI commands, as documented here (`https://docs.microsoft.com/en-us/cli/azure/keyvault/secret?view=azure-cli-latest#az-keyvault-secret-set`). On the other hand, in this case, since the data will be written in clear text in the code, it will be necessary to secure it well by giving read and write permissions only to authorized persons.

See also

- Documentation on the block data, `azurerm_key_vault_secret`, is available here: `https://www.terraform.io/docs/providers/azurerm/d/key_vault_secret.html`

Provisioning and configuring an Azure VM with Terraform

In this recipe, we will study a typical use case of Terraform in Azure in which we will provision and configure a VM in Azure using Terraform.

Getting ready

For this recipe, we don't need any special prerequisites. We will start the Terraform configuration from scratch. This recipe will only involve writing the Terraform configuration. We will explore the process of writing the Terraform configuration in different stages as we proceed with this recipe. As for the architecture in Azure, we have already built a network beforehand, which will contain this VM and which is made up of the following resources:

- A **virtual network (VNet)** called `VNET-DEMO`.
- Inside this VNet, a subnet named `Subnet1` is registered.

In addition, the VM that will be provisioned will have a public IP address so that it can be accessed publicly.

Note that, for this recipe, we will expose the VM with a public IP, but in production, it is not recommended to use a public IP; it is recommended to use a private endpoint as documented here: `https://learn.microsoft.com/azure/private-link/private-endpoint-overview`.

Finally, keeping the VM's password secret in the code, we protect it in an Azure Key Vault resource, as studied in the *Using Azure Key Vault with Terraform to protect secrets* recipe of this chapter.

The source code for this chapter is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/vm`.

How to do it...

Write the following Terraform configuration to provision a VM with Terraform:

1. The first resource to build is the resource group, with the help of the following code:

```
resource "azurerm_resource_group" "rg" {
  name     = "RG-VM"
  location = "EAST US"
}
```

2. Then, we write the following code to provision the public IP:

```
resource "azurerm_public_ip" "ip" {
  name                = "vmdemo-pip"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  allocation_method   = "Dynamic"
}
```

3. We continue by writing the code for the network interface:

```
data "azurerm_subnet" "subnet"{
  name = "Default1"
  resource_group_name = "RG_NETWORK"
  virtual_network_name = "VNET-DEMO"
}
```

```
resource "azurerm_network_interface" "nic" {
  name                = "vmdemo-nic"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location

  ip_configuration {
    name                          = "internal"
    subnet_id                     = data.azurerm_subnet.subnet.id
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id          = azurerm_public_ip.ip.id
  }
}
```

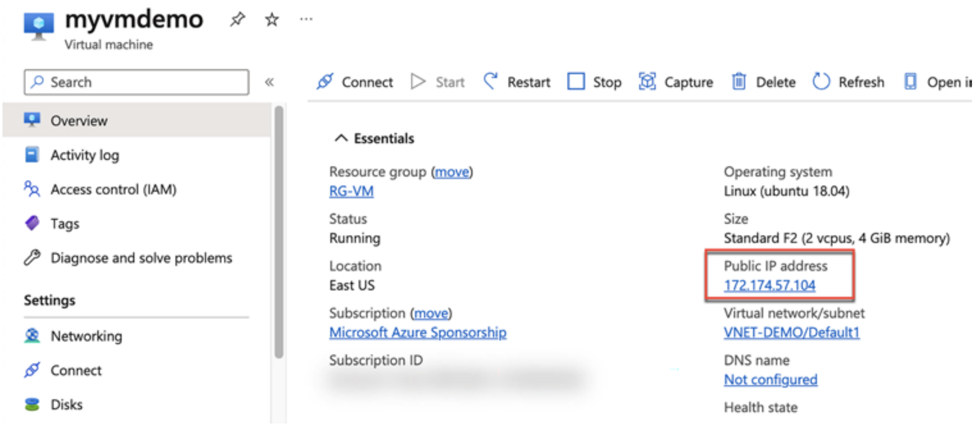
4. We get the VM password by using the `random_password` resource:

```
resource "random_password" "password" {
  length           = 16
  special          = true
  override_special = "_%@"
}
```

5. Finally, we write the code for the VM resource, as follows. (Here is an extract of the Terraform configuration, the complete code is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP08/vm/main.tf`):

```
resource "azurerm_linux_virtual_machine" "vm" {
  name                            = "myvmdemo"
...
  admin_username                  = "adminuser"
  admin_password                  =
random_password.password.result
  network_interface_ids = [azurerm_network_interface.nic.id]

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }
...

  provisioner "remote-exec" {
    inline = [
      "sudo apt update"
"sudo apt install nginx -y",
    ]
    connection {
      host     = self.public_ip_address
      user     = self.admin_username
      password = self.admin_password
    }
  }
}
```

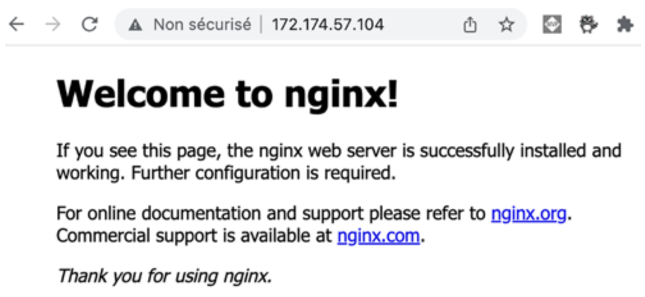
6. Finally, we set the four environment variables for the Terraform Azure authentication and we run the Terraform workflow with the terraform `init`, `plan`, and `apply` commands.
7. Go to the Azure portal on the created VM properties, and get the public IP value, as shown in the following image:



Resource group (move)
RG-VM
Status
Running
Location
East US
Subscription (move)
Microsoft Azure Sponsorship
Subscription ID
[redacted]
Operating system
Linux (ubuntu 18.04)
Size
Standard F2 (2 vcpus, 4 GiB memory)
Public IP address
172.174.57.104
Virtual network/subnet
VNET-DEMO/Default1
DNS name
Not configured
Health state

Figure 8.13: Getting the VM's public IP

8. Open a browser and navigate to `http://<PUBLIC_IP>`:



Non sécurisé | 172.174.57.104

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 8.14: nginx default website

We can see that we have access to the successfully installed nginx web server.

How it works...

In *Step 1*, we wrote the Terraform configuration that will create the resource group containing the VM. This step is optional because you can provision the VM in an existing resource group, and in this case, you can use the `azurerm_resource_group` block data, whose documentation is available here: `https://www.terraform.io/docs/providers/azurerm/d/resource_group.html`.

Then, in *Steps 2* and *3*, we wrote the Terraform configuration that provides the following:

- A public IP of the dynamic type, so that we don't have to set the IP address (this IP address will be the first free address of the subnet).
- The network interface of the VM that uses this IP address, which will register in the subnet that has already been created. To retrieve the subnet ID, we used an `azurerm_subnet` data source.

In *Step 4*, we use the `random_password` resource to generate a random password for the VM (refer to the *Generating passwords with Terraform* recipe in *Chapter 2*, *Writing Terraform Configurations*, for more details).

Finally, in *Step 5*, we write the code that will provision the VM. In this code, we have defined the following properties of the VM:

- Its name and size (which includes its RAM and CPU)
- The basic image used, which is an Ubuntu (Linux) image
- Authentication information for the VM with a login and a password (an SSH key can also be used but is not detailed in this recipe)

In this resource, we also added a `remote-exec` provisioner, which allows you to remotely execute commands or scripts directly on the VM that will be provisioned. The use of this provisioner will allow you to configure the VM for administration, security, or even middleware installation tasks. Here, in this recipe, we use the provisioner to install nginx using the command `apt install nginx`.

At the end of the execution of the Terraform commands, we open the browser at the public IP URI address and we get a default installed nginx website.

There's more...

The interesting and new aspect of this recipe is the addition of the `remote-exec` provisioner, which enables the configuration of the VM using commands or scripts. This method can be useful in performing the first steps of VM administration, such as opening firewall ports, creating users, and other basic tasks. Here, in our recipe, we used it to update the packages with the execution of the `apt update` command. However, this method requires that this VM is accessible from the computer running Terraform because it connects to the VM (SSH or WinRM) and executes the commands.

If you want to keep a real IaC, it is preferable to use an Code as configuration tool, such as Ansible, Puppet, Chef, or PowerShell DSC. And so, in the case of using Ansible to configure a Windows VM, the `remote-exec` provisioner can perfectly serve to authorize the WinRM SSL protocol on the VM because this port is the port used by Ansible to configure Windows machines.

Moreover, in Azure, you can also use a custom script VM extension, which is another alternative to configuring VMs using a script. In this case, you can provision this VM extension with Terraform using the `azurerm_virtual_machine_extension` resource, as explained in the following documentation: `https://www.terraform.io/docs/providers/azurerm/r/virtual_machine_extension.html`.

There can only be one custom script extension per VM. Therefore, you have to put all the configuration operations in a single script.

Apart from providing `remote-exec` and the VM extension, another solution is to use the `custom_data` property of the Terraform resource, `azurerm_virtual_machine`. Documentation pertaining to the `custom_data` property is available at `https://www.terraform.io/docs/providers/azurerm/r/linux_virtual_machine.html#custom_data`, and a complete code sample is available at `https://github.com/terraform-providers/terraform-provider-azurerm/blob/master/examples/virtual-machines/linux/custom-data/main.tf`.

Finally, by way of another alternative for VM configuration, we can also preconfigure the VM image with all the necessary software using **Packer**, which is another open source tool from HashiCorp and allows you to create your own VM image using JSON or HCL2 (as documented at `https://www.packer.io/guides/hcl`). Once this image is created, in the Terraform VM configuration, we will set the name of the image created by Packer instead of the image provided by the Marketplace (Azure or other cloud providers). For more information about Packer, read the following documentation: `https://www.packer.io/`.

The differences between these two solutions are:

- The `remote-exec` or custom script is useful if you want to install or configure software or components, but it doesn't guarantee the immutability of the OS configuration and its security hardening, because anyone who writes the Terraform configuration can insert a security hole in the VM. So you need to use it with care and control what is put into the Terraform configuration.
- As for Packer, it will be useful for ensuring OS immutability with its configuration and hardening. However, its use requires the construction of OS images and a frequent update pipeline.

See also

- Various tutorials and guides are available in the Azure documentation available here: `https://docs.microsoft.com/azure/developer/terraform/create-linux-virtual-machine-with-infrastructure`

Building Azure serverless infrastructure with Terraform

In the previous recipe, we studied the implementation of the Terraform configuration that allows the provisioning of an IaaS (that is, a VM) infrastructure in Azure.

In this recipe, we will stay in the same realm as the previous recipe, but this time, we will focus on writing the Terraform configuration that is used to provision a PaaS serverless infrastructure with the provisioning of an Azure App Service resource.

Getting ready

The purpose of this recipe is to provision and configure an Azure App Service resource of the web app type. In addition to provisioning, we will deploy an application in this web app at the same time as it is being provisioned using Terraform.

Most of the Terraform configuration needed for this recipe has already been studied in several recipes in this book. We will just study the Terraform configuration needed to deploy the application in this web app.

Regarding the application, it must be packaged in a ZIP file that is in the format `<appname>_<version>.zip`, such as `myapp_v0.1.1.zip`, and then we will upload this ZIP file in an Azure Blob Storage resource. This ZIP file can be uploaded either via Azure CLI, as indicated in this documentation, `https://docs.microsoft.com/en-us/cli/azure/storage/blob?view=azure-cli-latest#az-storage-blob-upload`, or via Terraform using the `azurerm_storage_blob` resource, whose documentation is available here: `https://www.terraform.io/docs/providers/azurerm/r/storage_blob.html`.

The Terraform configuration we will write in this recipe will use this ZIP file in a secure way. The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/webapp`.

How to do it…

Perform the following steps to provision a web app with Terraform:

1. In a new Terraform file, copy and paste the web app Terraform configuration from `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/sample-app`.
2. Inside this Terraform file, we add a new `azurerm_storage_account` data block with the following code:

```
data "azurerm_storage_account" "storagezip" {
  name                = "storappdemo"
  resource_group_name = "RG-storageApp"
}
```

3. Then, we add another `azurerm_storage_account_sas` data block to get a security token with the following extract code:

```
data "azurerm_storage_account_sas" "storage_sas" {
  connection_string = data.azurerm_storage_account.storagezip.
primary_connection_string
...
  services {
    blob  = true
...
  }
```

```
    start  = "2020-06-15"
    expiry = "2024-09-21"
    permissions {
      read     = true
      write    = false
     ...
    }
  }
```

The complete code of this block is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP08/webapp/main.tf`.

How it works...

In *Step 1*, we retrieved the Terraform configuration that allows a web app to be provisioned. In the following steps of the recipe, we will complete it in order to be able to deploy a web application directly in this web app, with Terraform, at the same time as its provisioning.

Then, in *Step 2*, we add the `azurerm_storage_account` data block, which will allow us to retrieve properties from the Storage Account that contains the ZIP file of the application. In *Step 3*, we add the `azurerm_storage_account_sas` data block, which will return a security token to the blob.

In this token, we indicate that the access will be read-only and that we only give access to the blob service.

Finally, in *Step 4*, we complete the `azurerm_linux_web_app` resource by adding in the application settings of the `WEBSITE_RUN_FROM_PACKAGE` key, which contains, by way of a value, the complete URL of the ZIP file and in which we concatenated the token key returned in the block.

There's more...

In this recipe, we have studied the possibility of provisioning an Azure web app and deploying it with Terraform. There are, however, several other ways to deploy this application in the web app, as explained in the documentation available at `https://docs.microsoft.com/en-us/azure/app-service/deploy-zip`.

In the *Building CI/CD pipelines for Terraform configuration in Azure Pipelines* recipe in *Chapter 13*, *Automating Terraform Execution in a CI/CD Pipelines*, we will learn how to automate this deployment in a CI/CD pipeline in Azure Pipelines.

See also

- Documentation pertaining to the `WEBSITE_RUN_FROM_PACKAGE` app setting of a web app is available here: `https://docs.microsoft.com/en-us/azure/app-service/deploy-run-package`
- Documentation pertaining to the `azurerm_storage_account_sas` block data is available here: `https://www.terraform.io/docs/providers/azurerm/d/storage_account_sas.html`
- Documentation pertaining to the Terraform resource, `azurerm_linux_web_app`, is available here: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/linux_web_app`

Generating a Terraform configuration for existing Azure infrastructure

When enterprises want to automate their processes and adopt IaC practices (for example, with Terraform), they face the challenge of how to generate code for infrastructure that is already provisioned.

Indeed, for new infrastructure, it is sufficient to write the corresponding Terraform configuration and then execute it to provision it. On the other hand, for resources that are already provisioned, depending on their number and configuration, it can be long and tedious to write all the Terraform configuration and then execute it to also have the corresponding Terraform state file. In addition, this execution of the Terraform configuration can have side effects on these resources.

As a partial answer to this problem, we have seen, in the *Importing existing resources* recipe from *Chapter 5*, *Managing Terraform State*, that we can use the `terraform import` command to import the configuration of already provisioned resources into the Terraform state file.

However, this command requires that, on the one hand, the corresponding Terraform configuration is already written because this command only updates the Terraform state file, and on the other, this command must be executed in order for each resource to be imported.

With this in mind, and having already had this request from many clients, I asked myself the question: Are there tools or scripts that can be used to generate Terraform configuration and its Terraform state file for resources already provisioned in Azure?

In this recipe, I'm going to share the results of my investigation with you using one of the Terraform configuration generation tools called **Microsoft Azure Export for Terraform** (`aztfexport`) maintained by Azure; the documentation is here: `https://learn.microsoft.com/azure/developer/terraform/azure-export-for-terraform/export-terraform-overview`.

> In the first edition of this book, we used a tool called **Terraformer** in this recipe, which is hosted in the GitHub repo of Google Cloud Platform, at `https://github.com/GoogleCloudPlatform/terraformer`.
>
> In this second edition, I updated this recipe with a specific tool for Azure providers.
>
> You can read the content of the first edition here: `https://subscription.packtpub.com/book/cloud-and-networking/9781800207554/6/ch06lvl1sec65/generating-a-terraform-configuration-for-existing-azure-infrastructure#_ga=2.119507852.480259339.1673192707-504938521.1673192707`.

Getting ready

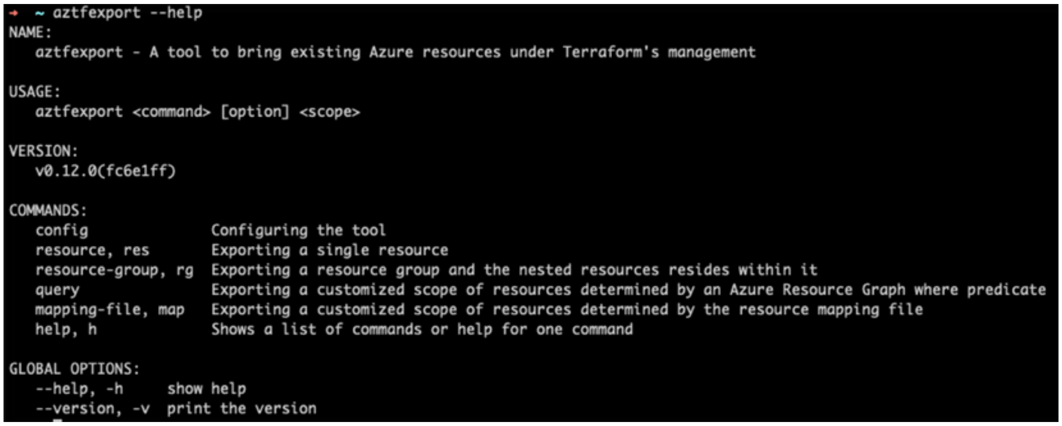
To use `aztfexport`, we need to download and install it. The installation procedure depends on the operating system that you use, and we can find all installation methods on this page: `https://github.com/Azure/aztfexport#install`.

For example, on Linux, we use the following script:

```
curl -sSL https://packages.microsoft.com/keys/microsoft.asc > /etc/apt/
trusted.gpg.d/microsoft.asc
ver=20.04 # or 22.04
apt-add-repository https://packages.microsoft.com/ubuntu/${ver}/prod
apt-get install aztfexport
```

This above script registers the Microsoft repository and installs the `aztfexport` tool.

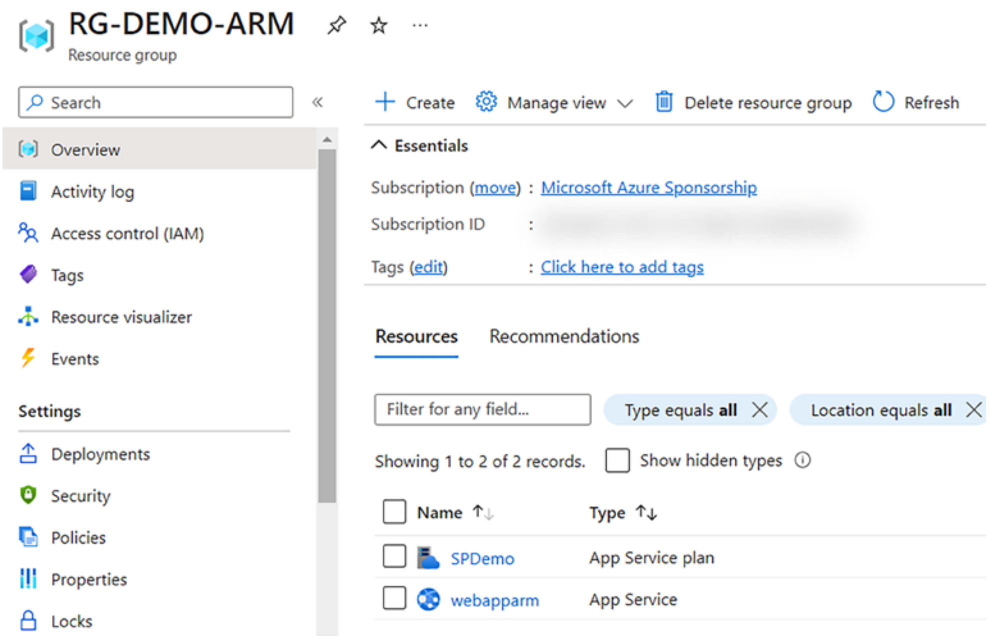
Once installed, its installation can be checked by executing the `aztfexport --help` command, and the list of `aztfexport` commands is displayed:



```
➜  ~ aztfexport --help
NAME:
   aztfexport - A tool to bring existing Azure resources under Terraform's management

USAGE:
   aztfexport <command> [option] <scope>

VERSION:
   v0.12.0(fc6e1ff)

COMMANDS:
   config              Configuring the tool
   resource, res       Exporting a single resource
   resource-group, rg  Exporting a resource group and the nested resources resides within it
   query               Exporting a customized scope of resources determined by an Azure Resource Graph where predicate
   mapping-file, map   Exporting a customized scope of resources determined by the resource mapping file
   help, h             Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --help, -h     show help
   --version, -v  print the version
```

Figure 8.15: aztfexport help command

The purpose of this recipe is to generate the Terraform configuration and the Terraform state file of an Azure infrastructure that is already created in Azure, which is composed of one resource group and, inside it, one service plan and one Azure App Service resource, as shown in the following screenshot:



RG-DEMO-ARM
Resource group

Subscription (move) : Microsoft Azure Sponsorship
Subscription ID :
Tags (edit) : Click here to add tags

Showing 1 to 2 of 2 records.

Name	Type
SPDemo	App Service plan
webapparm	App Service

Figure 8.16: Azure's existing resources to import

Once these commands have been run, we are ready to get started generating a Terraform configuration for our pre-existing Azure infrastructure.

Before running the `aztfexport` tool, we need to authenticate to Azure and select the subscription using the following commands:

```
az login
az account set -s <your subscription id>
```

Let's get started!

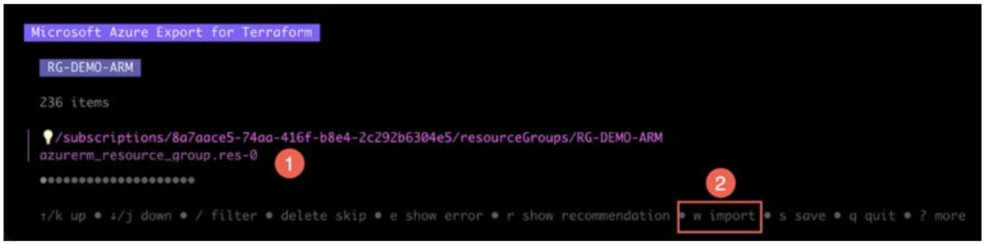
How to do it...

To generate a Terraform configuration using `aztfexport`, perform the following steps:

1. Inside your workstation, create a new folder named, for example, `azgenerated` that will contain the generated Terraform configuration.
2. Inside this new folder, in Terminal, run the `aztfexport rg` command:

   ```
   aztfexport rg "RG-DEMO-ARM"
   ```

3. The command analyzes all resources to import and asks you to choose the resources to import, as shown in the following screenshot:

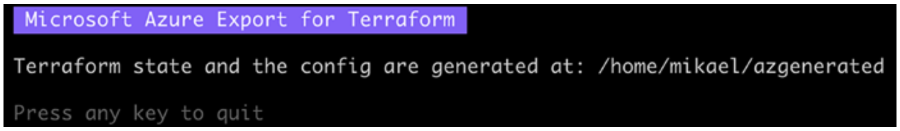
   

   ```
   Microsoft Azure Export for Terraform

     RG-DEMO-ARM

     236 items

   │ /subscriptions/8a7aace5-74aa-416f-b8e4-2c292b6304e5/resourceGroups/RG-DEMO-ARM
   │ azurerm_resource_group.res-0
     ••••••••••••••••••••

     ↑/k up • ↓/j down • / filter • delete skip • e show error • r show recommendation • w import • s save • q quit • ? more
   ```

   Figure 8.17: aztfexport import resources

   The command display all resources. We type "`w`" on the keyboard to import all resources in the selected resource – that is, the resource group.

4. The `aztfexport` tool imports the resources and displays a confirmation message when it is done, as is displayed in the following image:

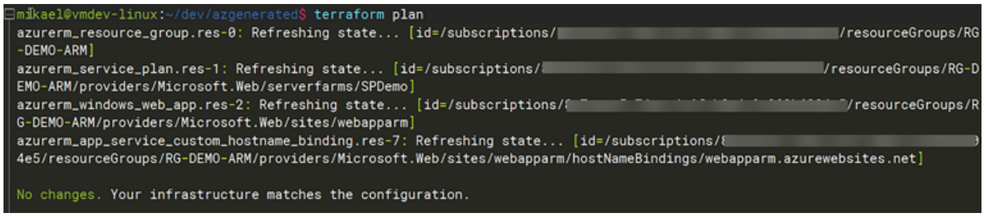
   

   ```
   Microsoft Azure Export for Terraform

   Terraform state and the config are generated at: /home/mikael/azgenerated

   Press any key to quit
   ```

   Figure 8.18: aztfexport import confirmation

We can then hit any keyboard key to close the tool.

5. Finally, in this azgenerated folder, we will test the configuration generated by running the basic Terraform workflow with the `terraform init` and `terraform plan` commands:



```
mikael@vmdev-linux:~/dev/azgenerated$ terraform plan
azurerm_resource_group.res-0: Refreshing state... [id=/subscriptions/[redacted]/resourceGroups/RG
-DEMO-ARM]
azurerm_service_plan.res-1: Refreshing state... [id=/subscriptions/[redacted]/resourceGroups/RG-D
EMO-ARM/providers/Microsoft.Web/serverfarms/SPDemo]
azurerm_windows_web_app.res-2: Refreshing state... [id=/subscriptions/[redacted]/resourceGroups/R
G-DEMO-ARM/providers/Microsoft.Web/sites/webapparm]
azurerm_app_service_custom_hostname_binding.res-7: Refreshing state... [id=/subscriptions/[redacted]
4e5/resourceGroups/RG-DEMO-ARM/providers/Microsoft.Web/sites/webapparm/hostNameBindings/webapparm.azurewebsites.net]

No changes. Your infrastructure matches the configuration.
```

Figure 8.19: terraform plan after resource import

If the output is generated successfully, we should see that the configuration generated does not apply any changes. It corresponds to our infrastructure exactly.

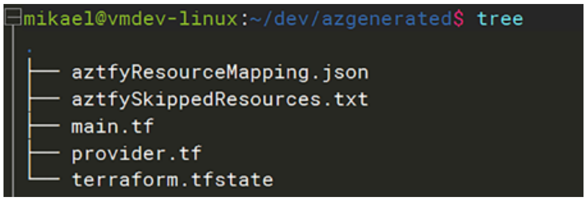
How it works...

In *Step 1*, we create a new folder that will contain the generated Terraform configuration.

In *Step 2*, we run the command `aztfexport rg <resource group>` to generate the Terraform configuration of the entire resource group resources.

In *Steps 3* and *4*, the tools propose several options, and we choose to import all resources inside this resource group.

Then, when the import is done, we can see the list of generated files inside the `azgenerated` folder:

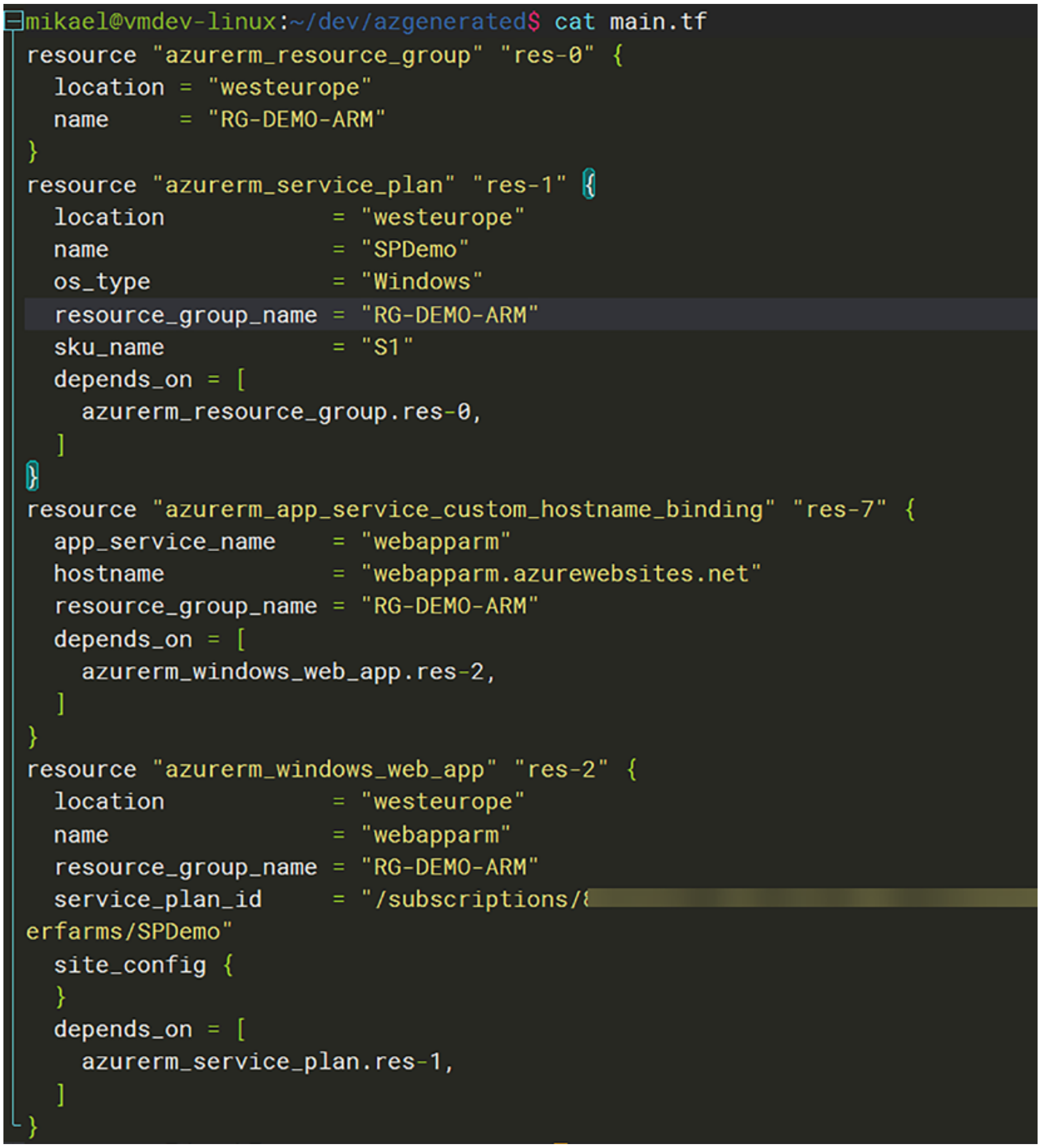


```
mikael@vmdev-linux:~/dev/azgenerated$ tree
.
├── aztfyResourceMapping.json
├── aztfySkippedResources.txt
├── main.tf
├── provider.tf
└── terraform.tfstate
```

Figure 8.20: aztfexport generated files

We can see `main.tf` and `provider.tf`, which contain the imported Terraform configuration, and `terraform.tfstate`, which is the Terraform state file of the imported resources (by default, the state file is generated locally).

The `main.tf` file contains the following Terraform configuration:

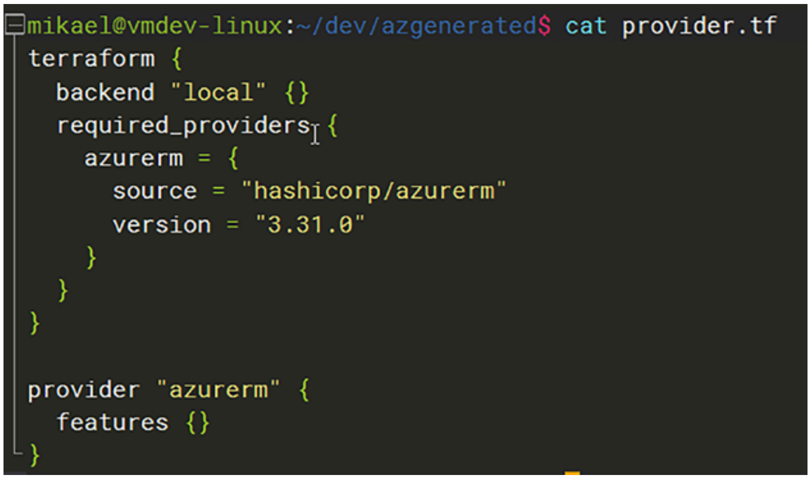


```
mikael@vmdev-linux:~/dev/azgenerated$ cat main.tf
resource "azurerm_resource_group" "res-0" {
  location = "westeurope"
  name     = "RG-DEMO-ARM"
}
resource "azurerm_service_plan" "res-1" {
  location            = "westeurope"
  name                = "SPDemo"
  os_type             = "Windows"
  resource_group_name = "RG-DEMO-ARM"
  sku_name            = "S1"
  depends_on = [
    azurerm_resource_group.res-0,
  ]
}
resource "azurerm_app_service_custom_hostname_binding" "res-7" {
  app_service_name    = "webapparm"
  hostname            = "webapparm.azurewebsites.net"
  resource_group_name = "RG-DEMO-ARM"
  depends_on = [
    azurerm_windows_web_app.res-2,
  ]
}
resource "azurerm_windows_web_app" "res-2" {
  location            = "westeurope"
  name                = "webapparm"
  resource_group_name = "RG-DEMO-ARM"
  service_plan_id     = "/subscriptions/[redacted]
erfarms/SPDemo"
  site_config {
  }
  depends_on = [
    azurerm_service_plan.res-1,
  ]
}
```

Figure 8.21: aztfexport generated Terraform configuration

We can see the configuration of the Azure resource group, the service plan, and the Azure web app.

The `provider.tf` file contains the following Terraform configuration:



```
mikael@vmdev-linux:~/dev/azgenerated$ cat provider.tf
terraform {
  backend "local" {}
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "3.31.0"
    }
  }
}

provider "azurerm" {
  features {}
}
```

Figure 8.21: aztfexport generated provider.tf

We can see the `terraform block` configuration and the `azurerm` provider configuration.

Finally, in *Step 5*, we verified that the generated code is equal to the provisioned resources by executing the `terraform init` and `plan` commands. During its execution, no changes will be applied. The Terraform configuration is well in line with our infrastructure.

There's more...

Here in this recipe, we used `aztfexport` with the optional `rg` to generate the Terraform configurations for all resources inside a resource group; however, we can also use `aztfexport` with the option to import the Terraform configuration for only one Azure resource.

This `aztfexport` tool contains other options to filter resources; read the documentation for more information here: `https://learn.microsoft.com/en-us/azure/developer/terraform/azure-export-for-terraform/export-terraform-overview`.

`aztfexport` is a tool to import and generate Terraform configuration for Azure infrastructure, there are other tools to import and generate Terraform configuration for more providers. Among these tools, there are:

- **Terraformer** (the documentation here: `https://github.com/GoogleCloudPlatform/terraformer`).
- **TerraCognita** (`https://github.com/cycloidio/terracognita/`), which still integrates a number of resources for Azure.

The problem with all these tools is that they must follow the evolution of the Terraform language and the evolution of different providers, which requires a lot of development and maintenance time.

See also

- The source code and the documentation of `aztfexport` are available here: `https://github.com/Azure/aztfexport`
- A blog post about `aztfy` (the legacy tool of `aztfexport`) is available here: `https://thomasthornton.cloud/2022/10/07/using-aztfy-to-import-existing-azure-resources-into-terraform/`
- A video showing how to use `aztfy` (the legacy tool of `aztfexport`) is available here: `https://www.youtube.com/watch?v=ADnTk3U22ew`

Enabling optional Azure features

One of the interesting points of the `azurerm` provider is the possibility to activate or deactivate some optional Azure features directly via the provider configuration.

To illustrate this provider feature, we will see in this recipe how to disable the deletion of an Azure resource group if this resource group contains resources.

Let's get started!

Getting ready

To complete this recipe, no requirements are needed.

We consider that the Terraform configuration used in this recipe is already written and it provisions a resource group. Inside this resource group, we create an Azure Storage Account manually (via the portal, Azure CLI, or another Terraform configuration).

The source code of this Terraform configuration is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP08/feature/main.tf`.

By default, without any other configuration, If we execute the command `terraform destroy` on our Terraform configuration, Terraform will not delete the resource group because it contains a Storage Account.

The goal of this recipe is to add configuration to enable the deletion of the resource group in such a scenario.

How to do it...

In our Terraform configuration, add the following configuration inside the provider configuration:

```
provider "azurerm" {
  features {
    resource_group {
      prevent_deletion_if_contains_resources = false
    }
  }
}
```

How it works...

In this recipe, we updated the `features` block by adding a `prevent_deletion_if_contains_resources` property with a `false` value.

By default, the value of this property is `true`, which prevents the deletion of the resource group that contains the resource.

Here, to enable the deletion of the resource group, we set the value of this property to `false`.

There's more...

To see the list of all properties of the `features` block, read the documentation here: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/guides/features-block`.

Estimating Azure cost of infrastructure using Infracost

By using IaC with Terraform, we see that we are able to rapidly provision a large-scale cloud infrastructure.

The question that is often asked by people who oversee finance is what the estimated cost of this infrastructure before being deployed is, but they also ask for an estimate of the delta of costs at each change of this infrastructure.

There is a tool called Infracost that allows you to estimate costs based on a Terraform configuration that involves a cloud infrastructure (Azure, AWS, or GCP).

Infracost (whose official website is `https://www.infracost.io/`) is a solution that consists of three components:

- The binary, which is open-source and free and which allows all the basic operations of resource pricing.
- Infracost Cloud, which is a SaaS that allows you to have integrated CI/CD features on `pull requests`, in addition to the features of the free version. Infracost also allows you to have policy integration with Jira, in addition to many other features that help with team integration. Information on the price of this version is available here: `https://www.infracost.io/pricing/`.
- The Infracost API, which connects to the different clouds to retrieve pricing information for the different cloud resources.

In this recipe, we will learn about the usage of Infracost with Terraform configuration that provisions an Azure resource.

> Note that the goal of Infracost is only to return cost estimation. It isn't the actual price, which can depend on other factors like network bandwidth, scalability, etc.

Let's get started!

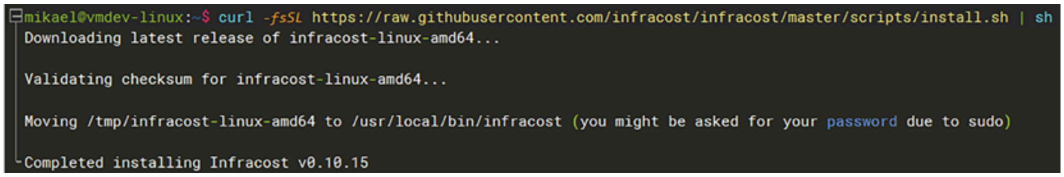
Getting ready

To complete this recipe, we will first install the Infracost binary by referring to the installation configuration that is available here: `https://www.infracost.io/docs/`.

For this recipe, we will install it on Linux using the following script:

```
curl -fsSL https://raw.githubusercontent.com/infracost/infracost/master/
scripts/install.sh | sh
```

The following image shows the output of the above script:



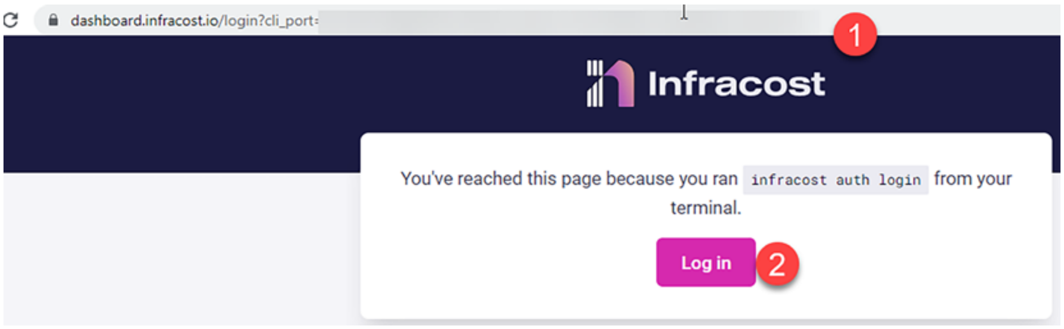
```
mikael@vmdev-linux:~$ curl -fsSL https://raw.githubusercontent.com/infracost/infracost/master/scripts/install.sh | sh
Downloading latest release of infracost-linux-amd64...

Validating checksum for infracost-linux-amd64...

Moving /tmp/infracost-linux-amd64 to /usr/local/bin/infracost (you might be asked for your password due to sudo)

Completed installing Infracost v0.10.15
```

Figure 8.23: Infracost install script

Then, after the Infracost installation, we need to log in to the Infracost system to generate an API key by executing the following command:

```
infracost auth login
```

This command opens a website in a web browser; then, on this web page, we click on the **Log in** button to authenticate ourselves anonymously, as we can see in the following image:


dashboard.infracost.io/login?cli_port=

You've reached this page because you ran `infracost auth login` from your terminal.

Figure 8.24: Infracost login

A confirmation is displayed on the browser page as well as in the command terminal.

In the command terminal, we also have information that the API key is generated as well, as shown in the following image:

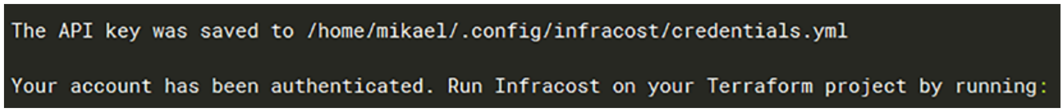

```
The API key was saved to /home/mikael/.config/infracost/credentials.yml

Your account has been authenticated. Run Infracost on your Terraform project by running:
```

Figure 8.25: Infracost store credentials

So, we are authenticated and an API key is generated. We can now run Infracost on a Terraform configuration.

The goal of this recipe is to estimate the cost of Azure infrastructure that is composed of:

- One resource group
- One VNet and subnet
- One virtual machine
- One service plan and one web app

All these resources are specified in the Terraform configuration whose source code is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/cost`.

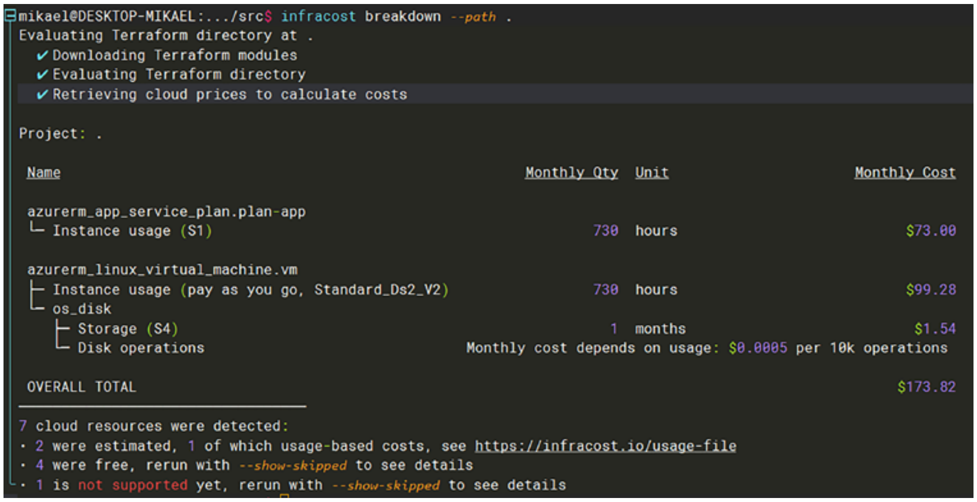
How to do it...

The first part is to perform the following steps to estimate the cost of these resources for the first provisioning:

1. The first step is to estimate the cost of these Azure resources before creating them. For this, inside the folder that contains the Terraform configuration, we run the following Infracost command:

   ```
   infracost breakdown --path .
   ```

   The following image shows the output of this command:

   ```
   mikael@DESKTOP-MIKAEL:.../src$ infracost breakdown --path .
   Evaluating Terraform directory at .
     ✔ Downloading Terraform modules
     ✔ Evaluating Terraform directory
     ✔ Retrieving cloud prices to calculate costs

   Project: .

    Name                                                      Monthly Qty  Unit              Monthly Cost

    azurerm_app_service_plan.plan-app
    └─ Instance usage (S1)                                            730  hours                   $73.00

    azurerm_linux_virtual_machine.vm
    ├─ Instance usage (pay as you go, Standard_Ds2_V2)                730  hours                   $99.28
    └─ os_disk
       ├─ Storage (S4)                                                  1  months                   $1.54
       └─ Disk operations                          Monthly cost depends on usage: $0.0005 per 10k operations

    OVERALL TOTAL                                                                                 $173.82
   ──────────────────────────────────
   7 cloud resources were detected:
   • 2 were estimated, 1 of which usage-based costs, see https://infracost.io/usage-file
   • 4 were free, rerun with --show-skipped to see details
   • 1 is not supported yet, rerun with --show-skipped to see details
   ```

   Figure 8.26: Infracost estimation cost

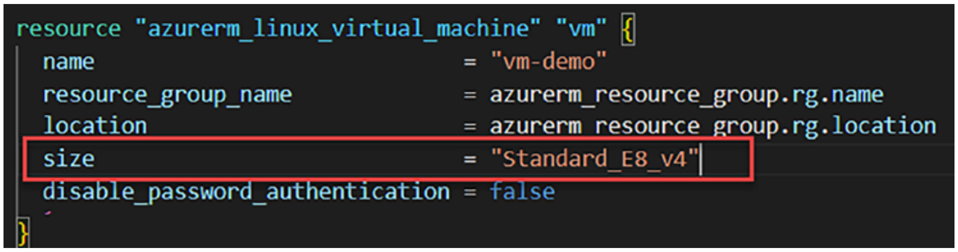
   According to these estimates, we will pay $173.82/month for the service plan and the virtual machine and its OS disk.

2. Then, we run the Terraform workflow to create these resources using the commands `terraform init`, `plan`, and `apply`.
3. Finally, we will run the following Infracost command to generate a JSON file with the preceding cost estimation:

   ```
   infracost breakdown --path . --format json --out-file infracost-
   base.json
   ```

In the second part, we perform the following steps to update our Terraform configuration and get the cost estimation only on the cost differential:

1. Update the Terraform configuration by changing the VM size to `Standard_E8_v4` (instead of `Standard_DS2_v2`), as shown in the Terraform configuration extract below:



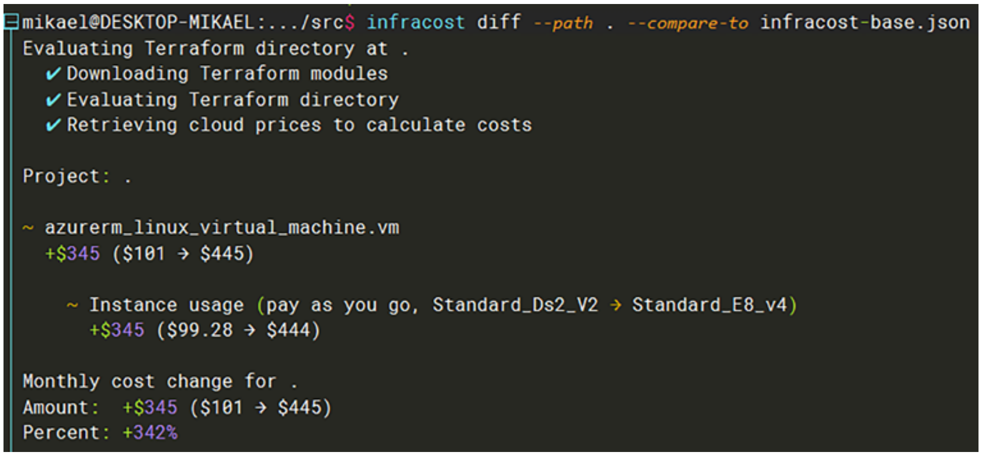
```
resource "azurerm_linux_virtual_machine" "vm" {
  name                            = "vm-demo"
  resource_group_name             = azurerm_resource_group.rg.name
  location                        = azurerm_resource_group.rg.location
  size                            = "Standard_E8_v4"
  disable_password_authentication = false
}
```

Figure 8.27: Change VM size

2. Then, we run the following command to have an estimate of the additional (or lower) costs that will be applied when we have executed this code change:

```
infracost diff --path . --compare-to infracost-base.json
```

The following image shows the result of its execution:



```
mikael@DESKTOP-MIKAEL:.../src$ infracost diff --path . --compare-to infracost-base.json
Evaluating Terraform directory at .
  ✔ Downloading Terraform modules
  ✔ Evaluating Terraform directory
  ✔ Retrieving cloud prices to calculate costs

Project: .

~ azurerm_linux_virtual_machine.vm
  +$345 ($101 → $445)

    ~ Instance usage (pay as you go, Standard_Ds2_V2 → Standard_E8_v4)
      +$345 ($99.28 → $444)

Monthly cost change for .
Amount:  +$345 ($101 → $445)
Percent: +342%
```

Figure 8.28: Infracost diff estimation

We can see that we will pay between $101 to $444 per month.

3. Finally, if we agree with this cost, we run the Terraform workflow to create these resources using the commands `terraform init`, `plan`, and `apply`.

How it works…

In the first part of this recipe, we perform a first cost estimation of Azure resources before creating them.

The used Infracost commands are:

```
infracost breakdown --path .
```

The `breakdown` option is to get the entire cost estimation of the infrastructure, and the `path` option targets the folder that contains the Terraform configuration to analyze.

Then we execute the command `infracost breakdown --path . --format json --out-file infracost-base.json` to export the cost estimation to an `infracost-base.json` file. We will use this file in the second part of this recipe.

In the second part of this recipe, we continue to use Infracost to estimate the cost difference after updating the Terraform configuration.

For this, we update the Terraform configuration by changing the VM size and we run the `infracost` command:

```
infracost diff --path . --compare-to infracost-base.json
```

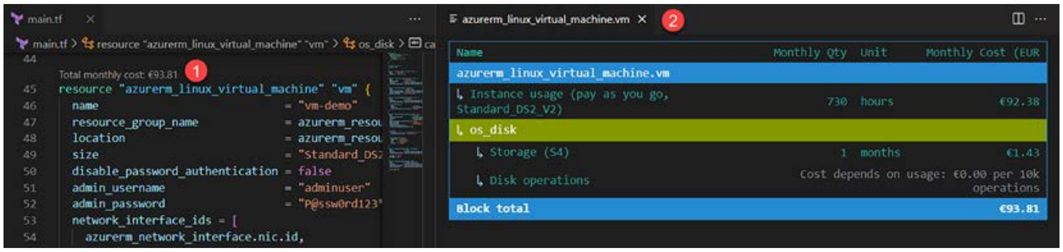
The `diff` option is used to perform a differential cost operation and the `compare-to` option specifies the JSON file to compare the cost, so we use the generated JSON file in the first part of the recipe.

There's more…

In this recipe, we learned about the basic usage of Infracost using the CLI. We can also use Infracost directly in VS Code with the Infracost extension available here: `https://marketplace.visualstudio.com/items?itemName=Infracost.infracost`.

With this extension, we can estimate resource costs at the same time as we write the Terraform configuration.

The following image shows the Infracost integration in VS Code:


main.tf
main.tf > resource "azurerm_linux_virtual_machine" "vm" > os_disk > ca
1
azurerm_linux_virtual_machine.vm
2

```
Total monthly cost: €93.81
resource "azurerm_linux_virtual_machine" "vm" {
  name                            = "vm-demo"
  resource_group_name             = azurerm_resou
  location                        = azurerm_resou
  size                            = "Standard_DS2
  disable_password_authentication = false
  admin_username                  = "adminuser"
  admin_password                  = "P@ssw0rd123"
  network_interface_ids = [
    azurerm_network_interface.nic.id,
```

```
Name                                          Monthly Qty  Unit     Monthly Cost (EUR
azurerm_linux_virtual_machine.vm
↳ Instance usage (pay as you go,
Standard_DS2_V2)                                      730  hours               €92.38
↳ os_disk
   ↳ Storage (S4)                                       1  months               €1.43
   ↳ Disk operations                  Cost depends on usage: €0.00 per 10k
                                                                          operations
Block total                                                                   €93.81
```

Figure 8.29: Infracost estimation in VS Code

When we click on the resource cost [**1**], a window opens in VS Code with the cost details for this resource [**2**].

Infracost can also be integrated into a CI/CD pipeline; for more information, read the documentation here: `https://www.infracost.io/docs/integrations/cicd/`.

In addition, advanced Infracost features are available in the Infracost Cloud solution. For more information, read the documentation here: `https://www.infracost.io/docs/infracost_cloud/get_started/`.

See also

- The documentation of Infracost is available here: `https://www.infracost.io/docs/`
- The Infracost blog is available here: `https://www.infracost.io/blog/`

Using the AzApi Terraform provider

The `azurerm` provider is an open source project that is maintained by HashiCorp, Microsoft, and by the community.

Its evolution depends on multiple factors like issue triage, maintenance work, and the Go Azure SDK version that is maintained by another team.

With all this, preview features of Azure are not added in real time in the `azurerm` provider.

So, if we want to use the Azure preview feature or another feature that might not yet be implemented in the `azurerm` provider, we can integrate it in the Terraform configuration in several ways, as we already learned:

- Use a `null_resource` resource that calls the `az cli` command, which we learned about previously in this chapter in the *Executing Azure CLI commands in Terraform* recipe.

- Use an **ARM template** deployed with a resource, which we learned about previously in this chapter in the *Executing ARM templates in Terraform* recipe

Since April 2022, we can use the new `AzApi` Terraform provider to provision Azure preview features.

In this recipe, we will learn how to use the `AzApi` Terraform provider.

Let's get started!

Getting ready

The goal of this recipe is to provision an Azure Storage Account using the `azurerm` provider and activate the STFP feature on this storage using the `AzApi` provider. The documentation is available here: `https://registry.terraform.io/providers/Azure/azapi/latest/docs`.

> In this scenario, the SFTP feature is shown as a preview feature. Note that it is no longer considered as a preview and has now been integrated into the `azurerm` provider. You can read more about this here: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/storage_account#sftp_enabled`.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/azapi`.

How to do it…

To activate SFTP on a storage account using the `AzApi` provider, perform the following steps:

1. First, in `main.tf`, we write the Terraform configuration to create the Azure resource group and Azure Storage using the following Terraform configuration:

```
terraform {
  required_version = "~> 1.1"
  required_providers {
    azurerm = {
      version = "~> 3.35"
    }
  }
}
provider "azurerm" {
  features {}
}
resource "azurerm_resource_group" "rg" {
    name      = "rg-demo-azapi"
    location  = "westeurope"
}

resource "azurerm_storage_account" "storage" {
    name                          = "accountsftpdemo"
    location                      = azurerm_resource_group.rg.location
    resource_group_name           = azurerm_resource_group.rg.name
    account_tier                  = "Standard"
    account_replication_type      = "LRS"
    min_tls_version               = "TLS1_2"
    is_hns_enabled                = true
}
```

2. Then, we complete the provider configuration to use the `AzApi` provider:

```
terraform {
  required_version = "~> 1.1"
  required_providers {
    azurerm = {
      version = "~> 3.35"
    }
    azapi = {
      source = "Azure/azapi"
      version = "1.1.0"
    }
  }
}
```

3. We add the configuration to enable the `sftp` feature to this Azure account:

```
resource "azapi_update_resource" "sftp_azpi_sftp" {
  type        = "Microsoft.Storage/storageAccounts@2021-09-01"
  resource_id = azurerm_storage_account.storage.id

  body = jsonencode({
    properties = {
      isSftpEnabled = true
    }
  })

    response_export_values = ["*"]
}
```

4. Finally, we run the Terraform workflow with the `init`, `plan`, and `apply` commands.

How it works...

In *Step 1*, we write the Terraform configuration to create an Azure resource group and an Azure Storage Account using the `azurerm` provider.

Then, in *Step 2*, we update the `terraform` block to specify the use of version 1.1.0 of the `Azure/azapi` provider.

In *Step 3*, we use the Terraform resource `azapi_update_resource` of the `AzApi` provider that provides the feature to add, update, enable, or disable properties of the existing Azure resource (here, this existing resource is the Storage Account specified with the property `resource_id`).

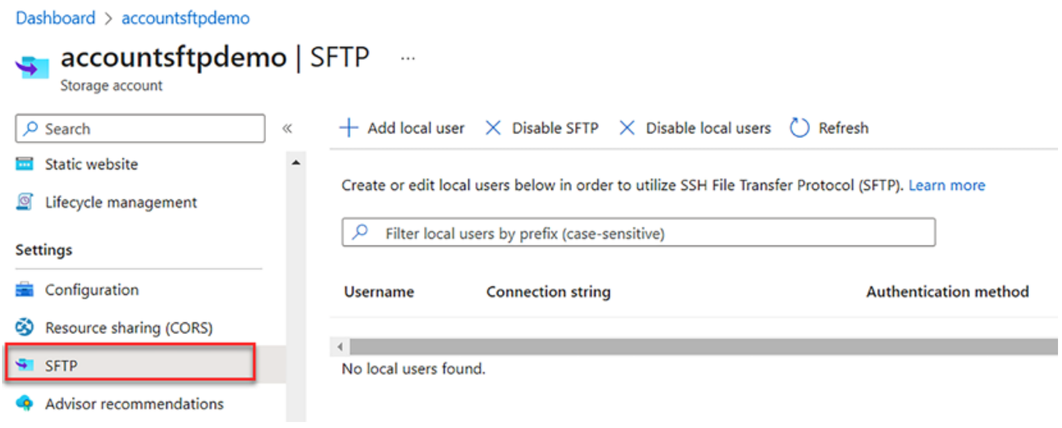
The `AzApi` provider calls the Azure REST API, so we need to pass it the API URL endpoint and the `body` payload.

Here, we set the `Azure API URL` in the type property, with the value: `Microsoft.Storage/storageAccounts@2021-09-01` and the `body` payload in JSON-encoded format with the following code:

```
body = jsonencode({
    properties = {
      isSftpEnabled = true
    }
  })
```

In the above `body`, we enabled the SFTP feature for the Storage Account. We also specify the explicit dependency with the Storage Account.

At the end, we run `init`, `plan`, and `apply` to provision this Storage Account, and after `apply`, we can check **SFTP** on the Azure portal, as in the following screenshot:



Create or edit local users below in order to utilize SSH File Transfer Protocol (SFTP). Learn more

Username	Connection string	Authentication method

No local users found.

Figure 8.30: Azure Storage SFTP enabled

We can see that the SFTP feature is enabled on the Storage Account; all that's left is to configure the list of local users.

There's more...

Here in this recipe, we learned how to use the `AzApi` Terraform provider to enable/disable a feature by changing a property in an existing resource. We can also provision a complete resource using the `azapi_resource` of the `azapi` provider (read the documentation here: `https://registry.terraform.io/providers/Azure/azapi/latest/docs/resources/azapi_resource`).

The advantages of using the `AzApi` provider are:

- The provider uses the Terraform state lifecycle
- It's possible to update an existing resource
- The provider uses the Azure REST API without the Go SDK
- No need to use an external script (like a JSON file) or tools (like `azcli`)

The disadvantage is:

- We need to explore the Azure API specification to understand the properties that need to be set in the `azapi_update_resource` resource's body property

See also

- The documentation of the `AzApi` provider is here: `https://registry.terraform.io/providers/Azure/azapi/latest/docs`
- Below is a list of some resources for the `AzApi` provider:
    - `https://learn.microsoft.com/en-us/azure/developer/terraform/get-started-azapi-resource`
    - `https://www.redeploy.com/post/day-zero-terraform-deployments-with-azapi`
    - `https://build5nines.com/azapi-terraform-provider-introduction-to-working-with-azure-preview-resources/`
    - `https://www.youtube.com/watch?v=VOod_VNgdJk`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

9

Getting Starting to Provisioning AWS and GCP Infrastructure Using Terraform

We have mentioned Azure a lot in previous chapters. Originally, we did so to explain Terraform configuration examples on concrete business cases. However, as of *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*, we have instead discussed Azure in the context of using Terraform to provision resources in Azure, which is what we will continue to cover now.

In this chapter, we will learn a basic way of using Terraform for provisioning AWS and GCP infrastructure. For each of these cloud providers, we will learn about how to create credentials for Terraform, the provider configuration, and the protection of the state file in the remote backend on AWS S3 and in GCP Storage.

For GCP, we will also learn how to use the integrated shell called Google Cloud Shell to execute Terraform without any local machine configuration.

Note that while this chapter is the starting point for AWS and GCP, all of the Terraform configuration, language, and best practices already learned about in all other chapters can also be applied to AWS and GCP.

In this chapter, we will cover the following recipes:

- Getting started using Terraform for AWS
- Using an S3 backend in AWS

- Getting started using Terraform for GCP
- Using a **Google Cloud Storage** (**GCS**) backend in GCP
- Executing Terraform in Google Cloud Shell

Technical requirements

To complete the recipes of this chapter, we will need to have the following service subscriptions:

- For AWS, we need to have an AWS subscription. You can get a free tier subscription here: `https://aws.amazon.com/free/`
- For GCP, we need to have a GCP account. You can subscribe to it here `https://console.cloud.google.com/freetrial/` with free tier services.

Getting started using Terraform for AWS

In this recipe, we will learn the basic steps to use Terraform to build simple resources on AWS.

We will principally discuss authentication and then resource provisioning with Terraform.

Let's get started!

Getting ready

To complete this recipe, you will need to generate user keys (an ID and secret key) that have permissions to create resources. In AWS, user or service account management is done via **IAM** (**Identity and Access Management**) and it is used to authenticate the **aws** provider.

To generate user keys, perform the following steps:

1. Log in to the AWS console by clicking on the **Sign In to the Console** button at the top:

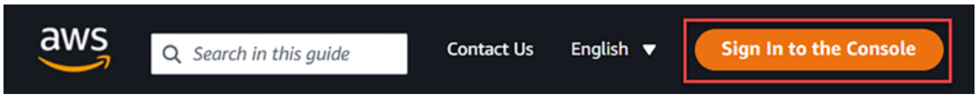


Figure 9.1: AWS console sign-in button

2. Then, in the top-right menu, click on your account name and, in the submenu, click on the **Security credentials** link, as shown in the following screenshot:

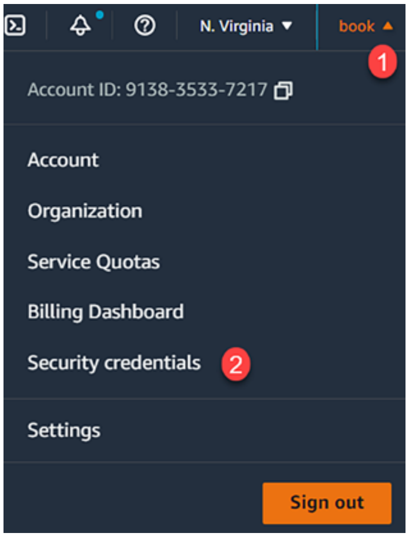
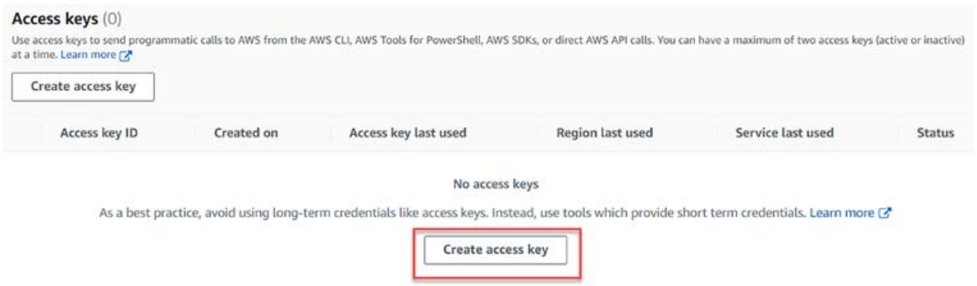


Figure 9.2: AWS Security credentials

3. On the page that opens, scroll down to the **Access Keys** section and click on the **Create access key** button.


Access keys (0)

Use access keys to send programmatic calls to AWS from the AWS CLI, AWS Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time. Learn more

Create access key

Access key ID	Created on	Access key last used	Region last used	Service last used	Status

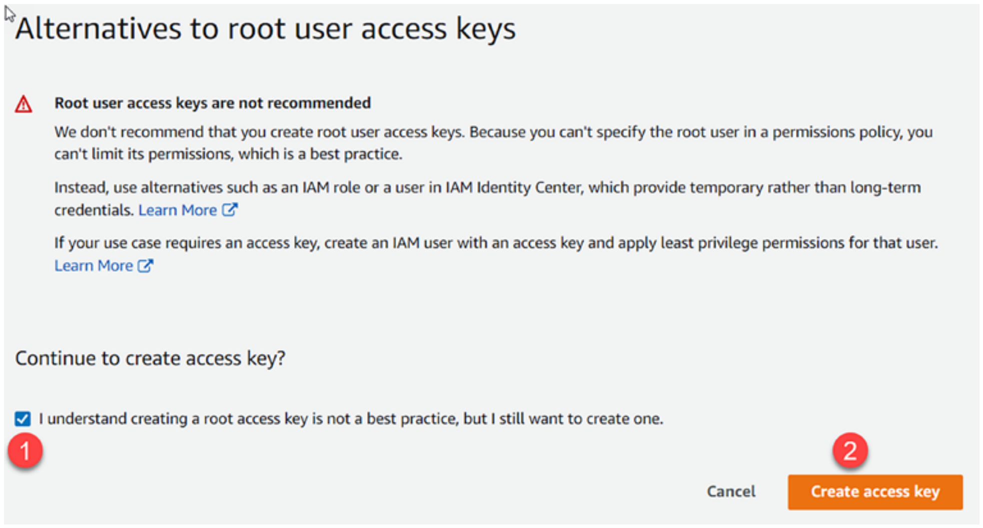
No access keys

As a best practice, avoid using long-term credentials like access keys. Instead, use tools which provide short term credentials. Learn more

Create access key

Figure 9.3: AWS Create access key

4. Then, check the checkbox to indicate that you understand that you are creating a root access key and click on **Create access key**.



Alternatives to root user access keys

Root user access keys are not recommended

We don't recommend that you create root user access keys. Because you can't specify the root user in a permissions policy, you can't limit its permissions, which is a best practice.

Instead, use alternatives such as an IAM role or a user in IAM Identity Center, which provide temporary rather than long-term credentials. Learn More

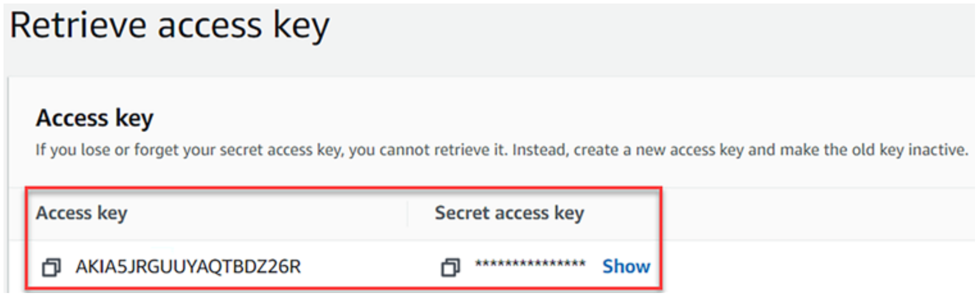
If your use case requires an access key, create an IAM user with an access key and apply least privilege permissions for that user. Learn More

Continue to create access key?

☑ I understand creating a root access key is not a best practice, but I still want to create one.

Figure 9.4: AWS Create access key confirmation

5. Finally, get the provided value of **Access key** and **Secret access key**.



Retrieve access key

Access key

If you lose or forget your secret access key, you cannot retrieve it. Instead, create a new access key and make the old key inactive.

Access key	Secret access key
AKIA5JRGUUYAQTBDZ26R	*************** Show

Figure 9.5: AWS access key information

Now that we have created our AWS account, we can use it with Terraform.

In this recipe, we will create an EC2 (virtual machine) resource using Terraform.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP09/aws`

How to do it…

Perform the following steps to provision an AWS resource:

1. Create a new file called `main.tf`. In this file, write the following Terraform configuration:

```
terraform {
  required_version = "~> 1.1"
  required_providers {
    aws = {
      version = "~> 3.27"
    }
  }
}

provider "aws" {
    region = "us-east-2"
}

resource "aws_instance" "my_ec2_instance" {
    ami = "ami-07c1207a9d40bc3bd"
    instance_type = "t2.micro"
}
```

2. Then, in the console, run the following commands:

```
export AWS_ACCESS_KEY_ID="<your access key ID>"
export AWS_SECRET_ACCESS_KEY="<your secret key> "
```

3. Finally, in the folder that contains the `main.tf` file, run the basic Terraform workflow with `init`, `plan`, and `apply` commands.

How it works...

In *Step 1*, we write the Terraform configuration that provisions an EC2 instance in AWS. In the configuration, we use the **aws** provider. Then, in the **provider** configuration block, we configure the provider to provision resources in the `us-east-2` region. Finally, in this configuration, we use the **aws_instance** resource to describe the virtual machine.

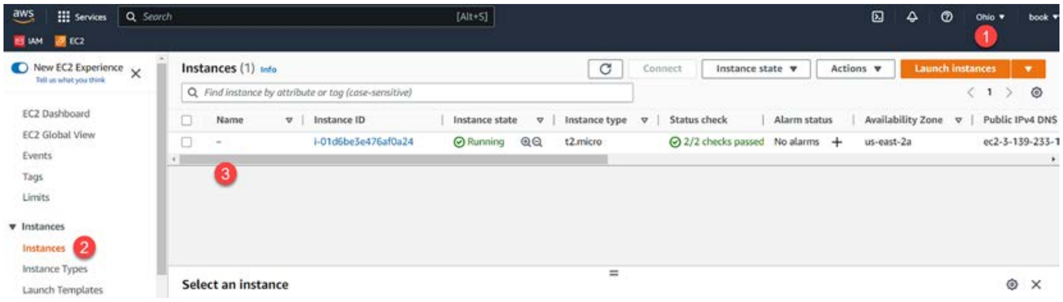
Note that, here, in the **aws_intance** resource, we use the **ami** property to have a simple configuration. For a more perfect configuration, use the **aws_ami** resource, and for more information, read the documentation here: `https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance#example-usage`.

In *Step 2*, we authenticate the **aws** provider using our **aws** credentials created previously, in the *Getting ready* section of this recipe.

For this, we set two variable environments, `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, with our access key ID and access key.

In *Step 3*, we run the Terraform workflow with `init`, `plan`, and `apply`.

After the execution of **apply**, we can check that the VM instance is created. In the AWS console, choose the **Ohio** region in the top menu, navigate under the EC2 instances, and check that your created instance is present and running, as shown in the following screenshot:



Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
-	i-01d6be3e476af0a24	Running	t2.micro	2/2 checks passed	No alarms	us-east-2a	ec2-3-139-233-1

Figure 9.6: AWS view instances list

Our EC2 instance is created and running.

There's more...

Here in this recipe, in *Step 2*, we use environment variables to set authentication credentials for the **aws** provider, which is the recommended method for security reasons.

We can also set the credentials directly in the Terraform configuration, in the provider configuration, as follows:

```
provider "aws" {
    region = "us-east-2"
    access_key = "<your access key ID>"
    secret_key = "<your secret key>"
}
```

You may be tempted to set the credentials directly in the Terraform configuration. This approach is, however, **not recommended** as it prevents the whole configuration from being easily reused, makes it hard to protect the keys, and increases the risk of those keys ending up in the VCS, in turn increasing the risk of leakage.

In development mode, we can also use the AWS CLI to authenticate the provider. For more information, read the documentation on AWS CLI configuration here: `https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html`.

Additionally, in this recipe, we use the root IAM access to authenticate the provider, which has full permissions on resources. In Infrastructure as Code best practices, it is recommended to use an IAM user with scope permissions (so, limited permissions). For this, create a user with programmatic access. For more information, read the documentation here: `https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html`.

See also

- The documentation of the **aws** Terraform provider is available here: `https://registry.terraform.io/providers/hashicorp/aws/latest/docs`

Using the S3 backend in AWS

In the previous recipe, we learned how to authenticate the *aws* Terraform provider, then how to use Terraform to provision resources on AWS.

In the *Protecting the state file in the Azure remote backend* recipe of *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*, we learned the importance of storing Terraform state in a backend and how to protect the Terraform state file in Azure Storage.

Now, using the same concept, in this recipe, we will learn how to store the state file in AWS using S3's bucket feature.

Let's get started!

Getting ready

To complete this recipe, there are no specific requirements.

We will create an S3 bucket manually in the AWS console and we will configure Terraform to use this S3 bucket as a Terraform state backend.

To create this S3 bucket, we will perform the following steps:

1. Sign in to the AWS console.

2. Create an S3 bucket by selecting **S3** in the **Services** menu (use the search text to find it).

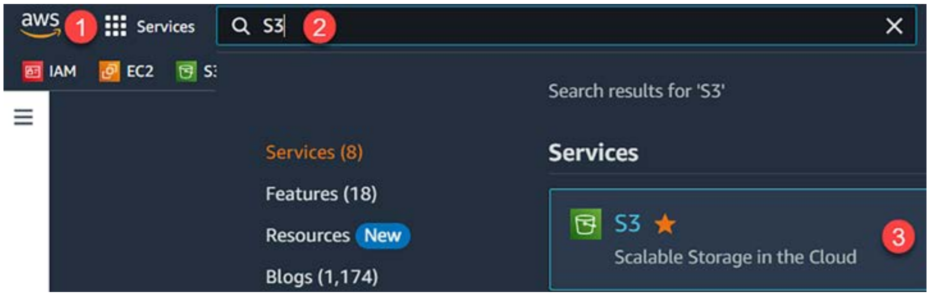
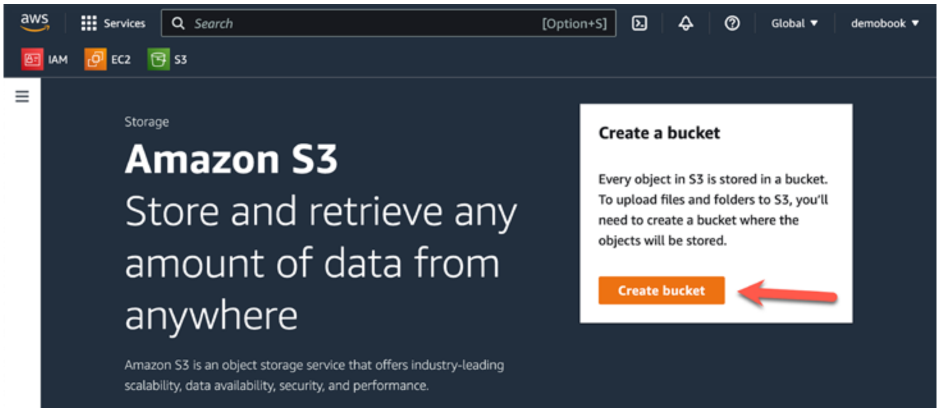


Figure 9.7: AWS – select the S3 service

3. Click on the **Create bucket** button.



Amazon S3

Store and retrieve any amount of data from anywhere

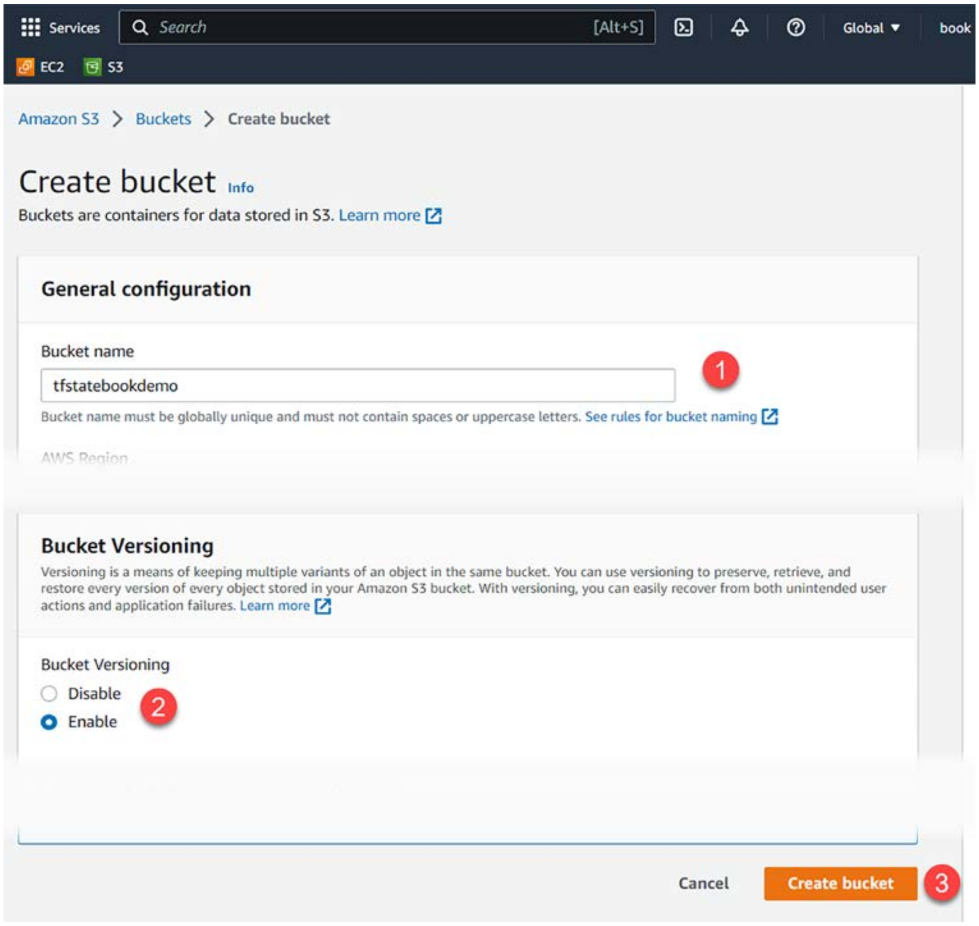
Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance.

Create a bucket

Every object in S3 is stored in a bucket. To upload files and folders to S3, you'll need to create a bucket where the objects will be stored.

Figure 9.8: AWS Create bucket button

4. Fill in the bucket form by entering the bucket name *tfstatebookdemo* (here the bucket name is given as example, you need to choose an unique name), activate the versioning, and click on the **Create bucket** button.



Create bucket

Buckets are containers for data stored in S3. Learn more

General configuration

Bucket name

tfstatebookdemo

Bucket name must be globally unique and must not contain spaces or uppercase letters. See rules for bucket naming

AWS Region

Bucket Versioning

Versioning is a means of keeping multiple variants of an object in the same bucket. You can use versioning to preserve, retrieve, and restore every version of every object stored in your Amazon S3 bucket. With versioning, you can easily recover from both unintended user actions and application failures. Learn more

Bucket Versioning

Disable [✓ Enable]

Figure 9.9: Filling in the S3 form

Now, in these steps of this recipe, we will learn how to use this S3 bucket to store our Terraform state.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP09/aws-state`.

How to do it...

To use this S3 bucket as Terraform remote backend, perform the following steps:

1. `main.tf` contains our Terraform configuration, complete the `terraform` block with the following configuration:

   ```
   terraform {
   …
     }
     backend "s3" {
       bucket = "tfstatebookdemo"
       key    = "terraform.tfstate"
       region = "us-east-2"
     }
   }
   ```

2. Then, set the two environment variables to authenticate the provider and the remote state configuration using the following commands:

   ```
   export AWS_ACCESS_KEY_ID="<your access key ID>"
   export AWS_SECRET_ACCESS_KEY="<your secret key> "
   ```

3. Finally, run the Terraform workflow with the `init`, `plan`, and `apply` commands.

How it works...

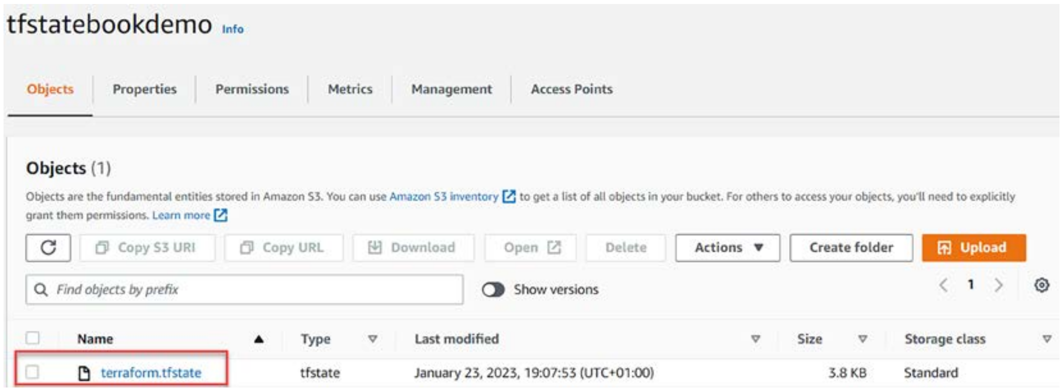
In the steps of this recipe, we add the configuration of the `s3` backend in the `terraform` block with the following properties:

- In the `bucket` property, we enter the bucket name `tfstatedemobook`.
- In the `key` property, we enter the name of the state object. Here, we configure `terraform.tfstate`.
- In the `region` property, we enter the region of the bucket.

Then we set the two **aws** credentials provider environment variables for the authentication.

Finally, we run the Terraform commands to apply changes to the Terraform configuration.

After the execution of Terraform, we can check the creation of the Terraform state object in the bucket. The following screenshot shows the content of the S3 bucket:



Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
terraform.tfstate	tfstate	January 23, 2023, 19:07:53 (UTC+01:00)	3.8 KB	Standard

Figure 9.10: AWS state file in S3 bucket

We can see `terraform.tfstate` in the bucket and the state file is now protected.

There's more...

In this recipe, we configure the S3 bucket with a basic option, that is, versioning. We can also add a DynamoDB table to add locks to S3 objects. For this AWS and Terraform configuration, read the following articles:

- `https://ksummersill.medium.com/setting-up-terraform-state-management-with-s3-bucket-and-dynamo-db-cfab238c1306`
- `https://www.golinuxcloud.com/configure-s3-bucket-as-terraform-backend/`

Additionally, we create the bucket by using the AWS console. We can also perform the same operations (including the DynamoDB table) using the AWS CLI. Read this article for more details: `https://skundunotes.com/2021/04/03/create-terraform-pre-requisites-for-aws-using-aws-cli-in-3-easy-steps/`.

It's also possible to create an AWS S3 bucket for the backend, but it is often avoided because it creates a "chicken and egg" problem, where it's not clear where to store the state for that bucket.

See also

- The documentation of the S3 remote backend is available here: `https://developer.hashicorp.com/terraform/language/settings/backends/s3`

Getting started using Terraform for GCP

In the previous recipes of this chapter, we learned how to use Terraform on AWS.

In this recipe, we will learn how to create a GCP service account and provision resources in GCP using Terraform.

Let's get started!

Getting ready

Before creating a resource in GCP with Terraform, we need to create a GCP service account that will have permission to provision resources in GCP.

To create a GCP service account using the console, perform the following steps:

1. In the **Services** menu, select the **IAM & Admin** menu and select the **Service Accounts** submenu.

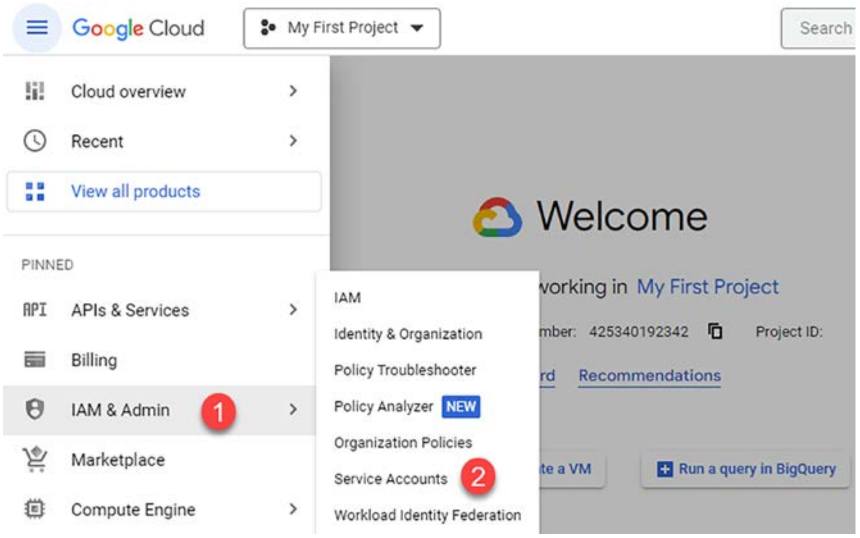
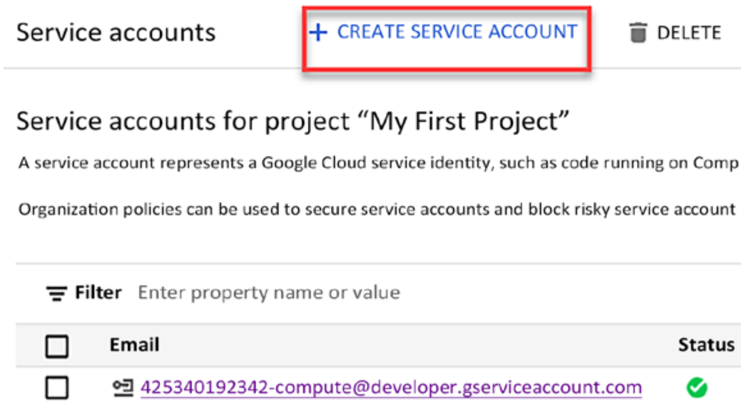


Figure 9.11: GCP Service Accounts menu

2. Then, click on the **CREATE SERVICE ACCOUNT** button.



Service accounts for project "My First Project"

A service account represents a Google Cloud service identity, such as code running on Comp

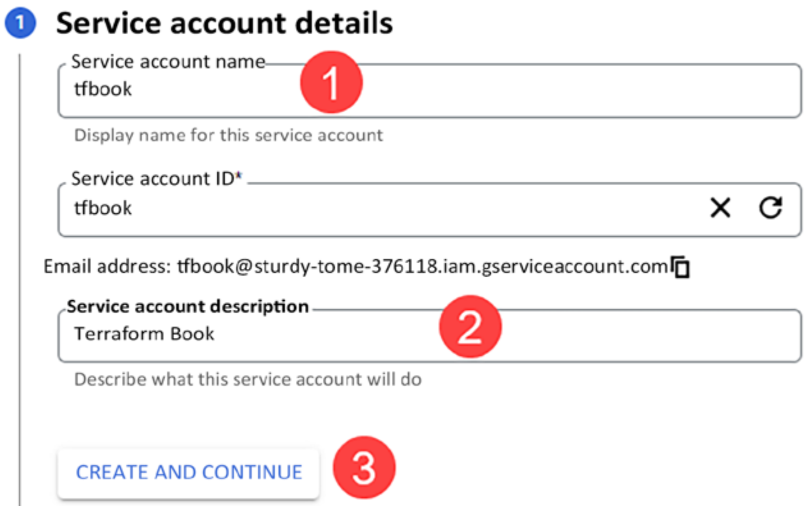
Organization policies can be used to secure service accounts and block risky service account

	Email	Status
☐	425340192342-compute@developer.gserviceaccount.com	✓

Figure 9.12: GCP CREATE SERVICE ACCOUNT button

3. Fill in the **Service accounts** form:

   a. Enter the service account name and description and click on the **CREATE AND CONTINUE** button.



Service account name: tfbook

Display name for this service account

Service account ID*: tfbook

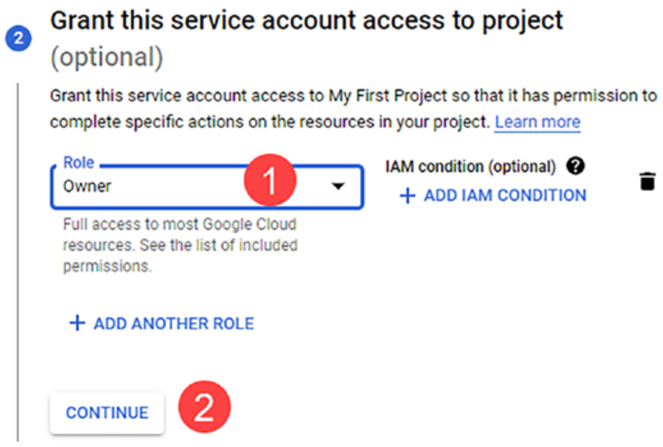
Email address: tfbook@sturdy-tome-376118.iam.gserviceaccount.com

Service account description: Terraform Book

Describe what this service account will do

Figure 9.13: GCP service account details form

b. Then, select **Basic** and the **Owner** role and click on the **CONTINUE** button.



Grant this service account access to project (optional)

Grant this service account access to My First Project so that it has permission to complete specific actions on the resources in your project. Learn more

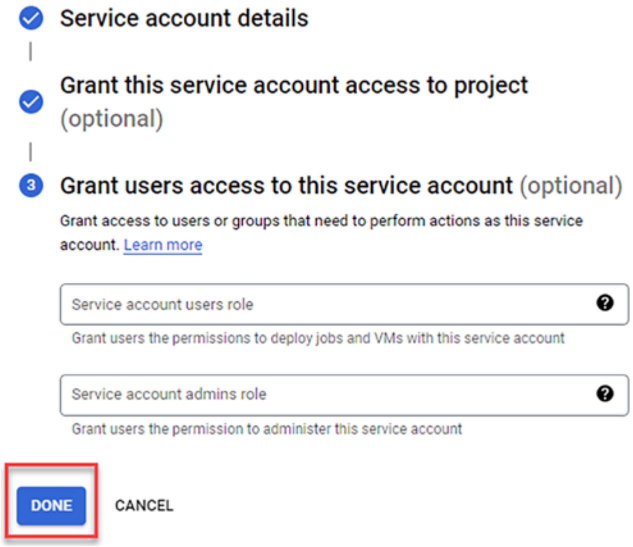
Role: Owner

Full access to most Google Cloud resources. See the list of included permissions.

IAM condition (optional)

Figure 9.14: GCP service account role

c. Finally, skip the optional *Step 3*, **Grant users access to this service account**, and click on the **DONE** button.



Service account details

Grant this service account access to project (optional)

Grant users access to this service account (optional)

Grant access to users or groups that need to perform actions as this service account. Learn more

Service account users role

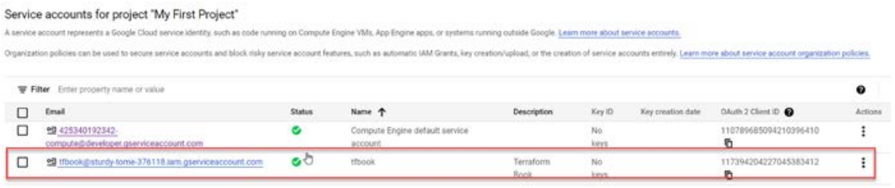
Grant users the permissions to deploy jobs and VMs with this service account

Service account admins role

Grant users the permission to administer this service account

Figure 9.15: Validating the service account

We can see our new service account in the service account list.



Service accounts for project "My First Project"

A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or systems running outside Google. Learn more about service accounts.

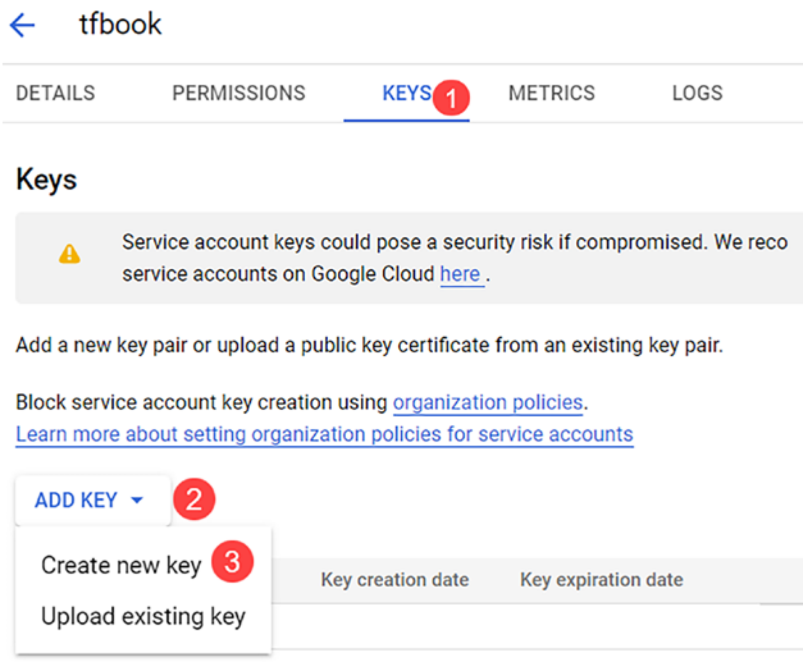
Organization policies can be used to secure service accounts and block risky service account features, such as automatic IAM Grants, key creation/upload, or the creation of service accounts entirely. Learn more about service account organization policies.

Filter Enter property name or value

Email	Status	Name ↑	Description	Key ID	Key creation date	OAuth 2 Client ID	Actions
425340192342-compute@developer.gserviceaccount.com	✓	Compute Engine default service account		No keys		110789685094210396410	⋮
tfbook@sturdy-tome-376118.iam.gserviceaccount.com	✓	tfbook	Terraform Book	No keys		117394204227045383412	⋮

Figure 9.16: GCP service account list

4. The last step is to generate and download a service account key. To do this, we navigate to the **KEYS** tab of the service account created and click on the **Create new key** option.



tfbook

DETAILS PERMISSIONS KEYS 1 METRICS LOGS

Keys

Service account keys could pose a security risk if compromised. We reco service accounts on Google Cloud here .

Add a new key pair or upload a public key certificate from an existing key pair.

Block service account key creation using organization policies.
Learn more about setting organization policies for service accounts

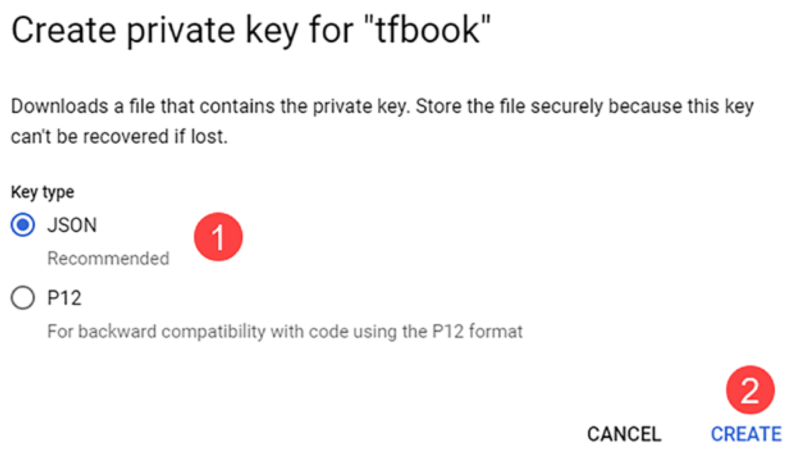
ADD KEY 2

Create new key 3
Upload existing key

Key creation date Key expiration date

Figure 9.17: Create new key for service account

5. In the pop-up window, choose the **JSON** format for the key and click on the **CREATE** button.



Create private key for "tfbook"

Downloads a file that contains the private key. Store the file securely because this key can't be recovered if lost.

Key type

● JSON
Recommended

○ P12
For backward compatibility with code using the P12 format

Figure 9.18: Exporting the JSON key

6. The JSON key file is generated, downloaded locally, and rename `gcp-key.json`. Keep this file as we will use it for authentication in the recipe steps.

We created a new GCP service account. Now we are ready to provision resources in GCP with Terraform.

In this recipe, we will create a compute instance (virtual machine) in the default VPC.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP09/gcp`.

How to do it...

To create a resource in GCP with Terraform, perform the following steps:

1. In the `main.tf` file, write the following Terraform configuration:

```
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
      version ~> "3.5.0"
    }
  }
}

provider "google" {
  region = "us-central1"
}
```

2. Then complete this file with the following configuration:

```
resource "google_compute_instance" "instance" {
  name         = "inst-demo-${random_string.random.result}"
  machine_type = "e2-micro"
  zone         = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-1804-lts"
    }
  }

  network_interface {
    network = "default"
  }
}
```

3. In the console, set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable with the path of the downloaded (in the requirements of this recipe) JSON file with credentials. Perform the following command:

```
export GOOGLE_APPLICATION_CREDENTIALS="$PWD/gcp-key.json"
```

4. Set the environment variable of the GCP project to use by executing the following command:

```
export GOOGLE_PROJECT="<your GCP project name>"
```

5. Finally, we run the basic Terraform workflow with the `init`, `plan`, and `apply` commands.

How it works...

In *Step 1*, we create a new `main.tf` and instantiate it with the configuration of the Terraform block for use with the `google` provider.

We also configure the `google` provider with the `google` project name and the region.

In *Step 2*, in this `main.tf`, we add the Terraform configuration to create the compute instance (VM) that will be created in the “default” VPC network.

Then, in *Step 3*, we authenticate the `google` provider using the service account credentials by setting a `GOOGLE_APPLICATION_CREDENTIALS` environment variable with the path of the located JSON file with the credentials of the service account.

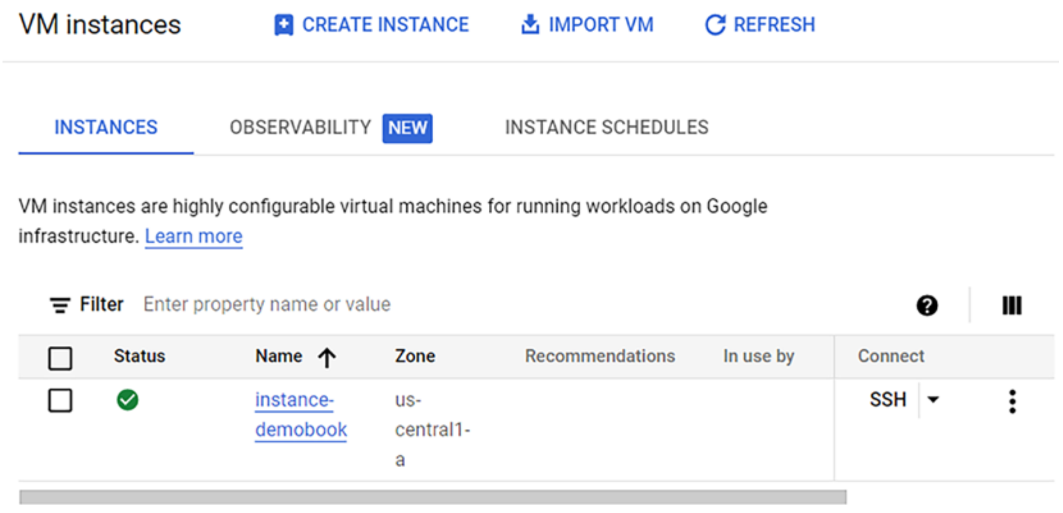
In *Step 4*, we set the `GOOGLE_PROJECT` environment variable with the name of the GCP project.

For more information about the provider configuration and specific environment variables, read the documentation here: `https://registry.terraform.io/providers/hashicorp/google/latest/docs/guides/provider_reference#project`

Finally, in *Step 5*, we run the basic Terraform workflow with the `init`, `plan`, and `apply` commands.

At the end of the execution, we can check the VM instance in the GCP dashboard.

The following screenshot shows the VM instance.



VM instances are highly configurable virtual machines for running workloads on Google infrastructure. Learn more

	Status	Name ↑	Zone	Recommendations	In use by	Connect
☐	✓	instance-demobook	us-central1-a			SSH

Figure 9.19: GCP instance status

The `instance-demobook` VM is created and running.

There's more...

For more information about basic Terraform provisioning for GCP, read the first chapter (free) of *Terraform for Google Cloud Essential Guide* here: `https://www.packtpub.com/product/terraform-for-google-cloud-essential-guide/9781804619629?_ga=2.12616958.85607785.1674469424-504938521.1673192707`

See also

- The documentation of Terraform on Google Cloud is available here: `https://cloud.google.com/docs/terraform`
- The documentation of the Terraform `google` provider is available here: `https://registry.terraform.io/providers/hashicorp/google/latest/docs`

Using a GCS backend in GCP

If you use Terraform to provision a resource on GCP, like for Azure and AWS, we need to store the Terraform state in the backend.

In this recipe, we will learn how to store a Terraform state file in a GCP storage bucket.

Let's get started!

Getting ready

There are no specific requirements to complete this recipe.

In this recipe, we will create a GCP storage bucket manually in the GCP dashboard and we will configure the Terraform block to use this GCP storage bucket as a state backend.

To create a GCP bucket, perform these steps:

1. Log in to your GCP console.

2. Go to the left menu and navigate to **Cloud Storage** > **Buckets**.

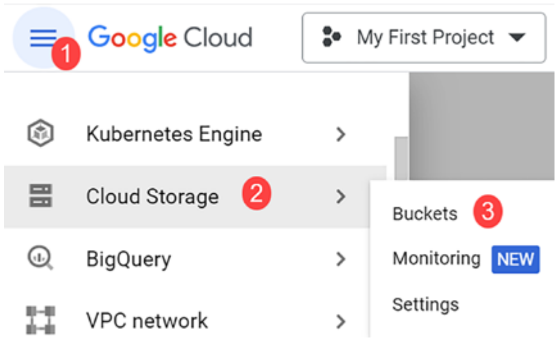


Figure 9.20: GCP Buckets menu

3. Click on the **CREATE** button.

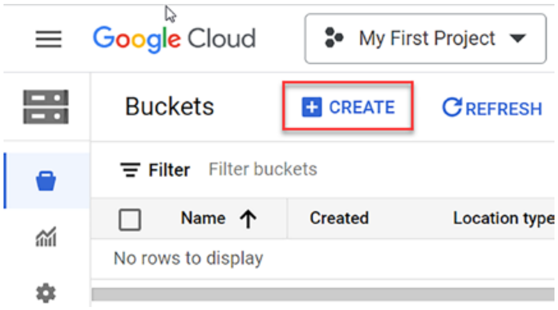
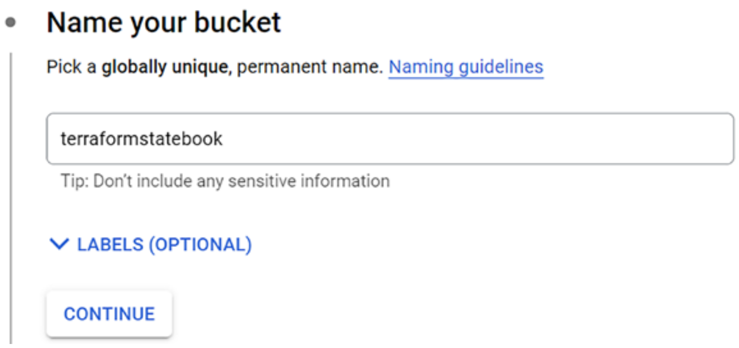


Figure 9.21: GCP bucket CREATE button

4. Fill in the form with the bucket name (here the bucket name is given as example, choose another unique name).



Name your bucket

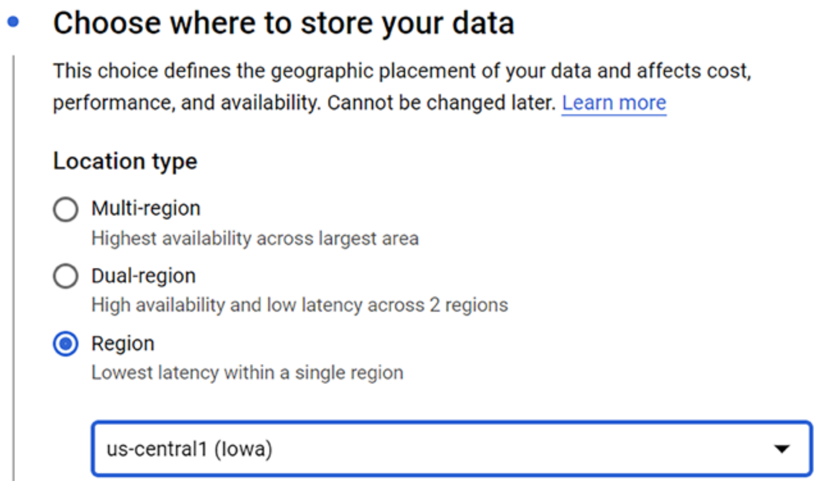
Pick a **globally unique**, permanent name. Naming guidelines

terraformstatebook

Tip: Don't include any sensitive information

Figure 9.22: GCP – fill in the name of the bucket

5. Then, choose **Region** (here, it is a single region):


Choose where to store your data

This choice defines the geographic placement of your data and affects cost, performance, and availability. Cannot be changed later. Learn more

Location type

( ) Multi-region
Highest availability across largest area

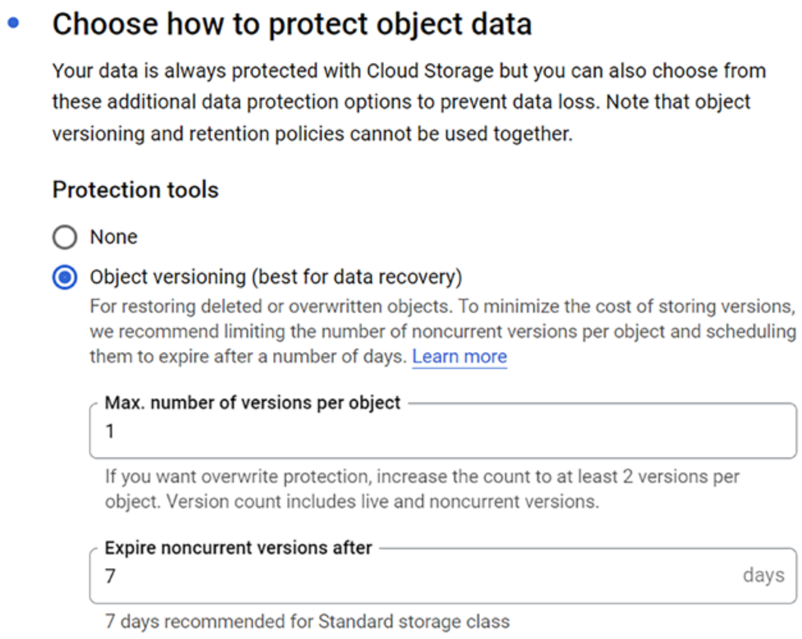
( ) Dual-region
High availability and low latency across 2 regions

[✓ Region]
Lowest latency within a single region

us-central1 (Iowa)

Figure 9.23: GCP bucket region

6. Then activate **Object versioning**.


Choose how to protect object data

Your data is always protected with Cloud Storage but you can also choose from these additional data protection options to prevent data loss. Note that object versioning and retention policies cannot be used together.

Protection tools

( ) None

[✓ Object versioning (best for data recovery)]
For restoring deleted or overwritten objects. To minimize the cost of storing versions, we recommend limiting the number of noncurrent versions per object and scheduling them to expire after a number of days. Learn more

Max. number of versions per object: 1
If you want overwrite protection, increase the count to at least 2 versions per object. Version count includes live and noncurrent versions.

Expire noncurrent versions after: 7 days
7 days recommended for Standard storage class

Figure 9.24: Activating object versioning

7. Finish by clicking on the **CREATE** button.

Now, in this recipe, we will learn how to use this bucket as a state backend.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP09/gcp-state`

How to do it…

To use this GCP bucket as a Terraform remote backend, perform the following steps:

1. On `main.tf`, which contains our Terraform configuration, complete the `terraform` block with the following configuration:

   ```
   terraform {
   …

     backend "gcs" {
       bucket = "<your created bucket name>"
       prefix = "tfstate"
     }
   }
   ```

2. Then, set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable with the path of the downloaded (in the requirements of this recipe) JSON file with credentials. Perform the following command:

   ```
   export GOOGLE_APPLICATION_CREDENTIALS="$PWD/gcp-key.json"
   ```

3. Finally, run the Terraform workflow with the `init`, `plan`, and `apply` commands.

How it works…

In the first part of this recipe, we create a GCP bucket, using the GCP console, that will store the Terraform state file.

For this bucket, we enter the name `tfstatebookdemo`, and out of the several options, we choose only one region and activate the versioning option.

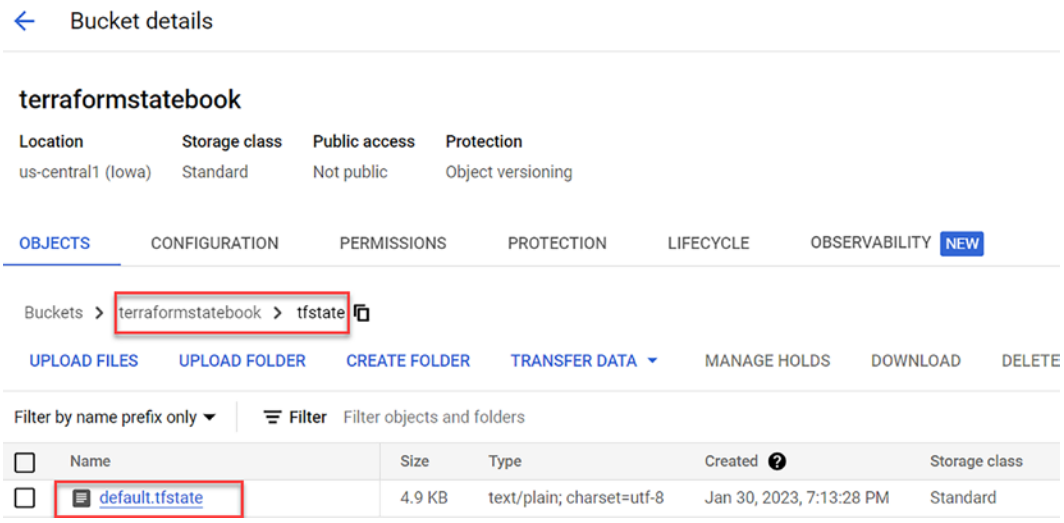
In the second part of this recipe, we add in the `terraform` block the configuration of this `gcs` backend with the following properties:

- In the `bucket` property, we enter the bucket name `tfstatedemobook`.
- In the `prefix` property, we enter the name of the state folder. Here, we configure `tfstate`.

Then we set the GOOGLE_APPLICATION_CREDENTIALS environment variable for provider authentication.

Finally, we run the Terraform init, plan, and apply commands to apply changes to the configuration.

After the execution of Terraform, we can check the creation of the state file as an object in the bucket. The following screenshot shows the content of the bucket:



Location	Storage class	Public access	Protection
us-central1 (Iowa)	Standard	Not public	Object versioning

Name	Size	Type	Created	Storage class
default.tfstate	4.9 KB	text/plain; charset=utf-8	Jan 30, 2023, 7:13:28 PM	Standard

Figure 9.25: GCP state file in bucket

The `default.state` state is created in the `tfstate` folder in the `terraformstatebook` bucket.

Just like with AWS, it's possible to create the GCP S3 bucket for the backend, but this is usually avoided because it's not clear where to store the state for that bucket.

See also

- The documentation of the gcs remote backend is available here: `https://developer.hashicorp.com/terraform/language/settings/backends/gcs`
- The Google Cloud documentation for Terraform with remote state is available here: `https://cloud.google.com/docs/terraform/resource-management/store-state`

Executing Terraform in GCP Cloud Shell

In the two previous recipes, we learned about the usage of Terraform configuration to provision GCP resources and deploy those resources from a local machine (or virtual machine).

Like in Azure, the GCP console contains a cloud shell that provides the possibility to run GCP commands and tools inside the browser.

In this recipe, we will learn how to run Terraform inside GCP Cloud Shell to provision GCP resources and use the GCP console-connected account to automatically authenticate the provider.

Let's get started!

Getting ready

To complete this recipe, you don't need to install any software: all software used is included in GCP Cloud Shell.

The only requirement is to have an active GCP account, which we can get for free here: `https://console.cloud.google.com/freetrial/`.

The goal of this recipe is to write a Terraform configuration and run it inside GCP Cloud Shell.

How to do it…

To run Terraform in GCP Cloud Shell, perform the following steps:

1. Launch GCP Cloud Shell by clicking on the top button.

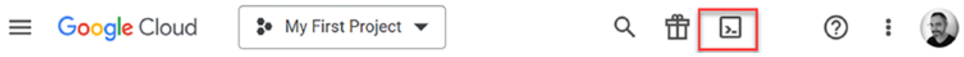
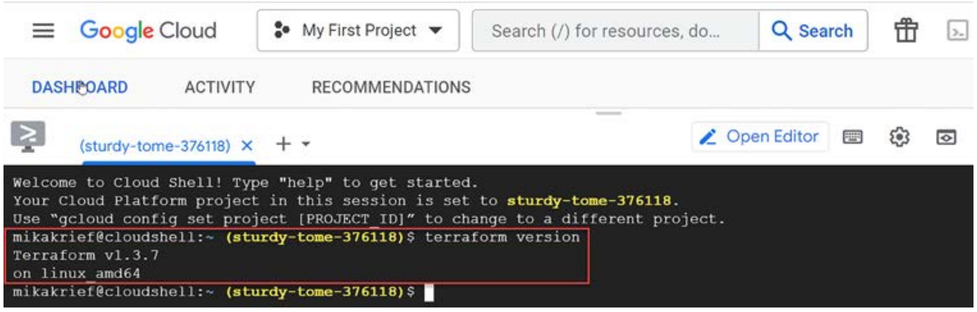


Figure 9.26: GCP Cloud Shell button

2. In the console that is open at the bottom, we can check the installed Terraform version by executing the `terraform version` command.



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to sturdy-tome-376118.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
mikakrief@cloudshell:~ (sturdy-tome-376118)$ terraform version
Terraform v1.3.7
on linux_amd64
mikakrief@cloudshell:~ (sturdy-tome-376118)$
```

Figure 9.27: GCP Terraform version in Cloud Shell

3. In the console, create a new folder, which will contain the Terraform configuration files, by executing the `mkdir terraform-demo` command.
4. Then, to write the Terraform configuration inside the `terraform-demo` folder, click on the **Open Editor** button.

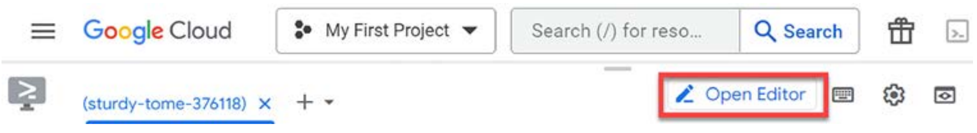


Figure 9.28: GCP Open Editor in Cloud Shell

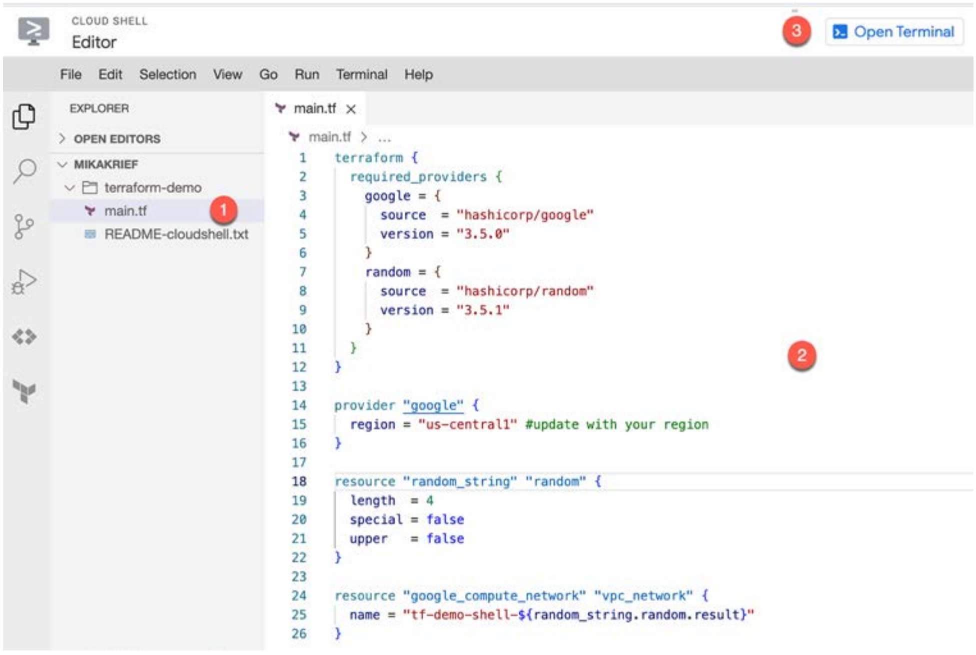
5. In the editor, in the `terraform-demo` folder, create a new `main.tf` file and copy-paste in this Terraform configuration:

```
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = "3.5.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.5.1"
    }
  }
}

provider "google" {
  region = "us-central1"
}

resource "random_string" "random" {
  length  = 4
  special = false
  upper   = false
}

resource "google_compute_network" "vpc_network" {
  name = "tf-demo-shell-${random_string.random.result}"
}
```

This code is also available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP09/gcp-shell/main.tf`

6. Click on the **Open Terminal** button.

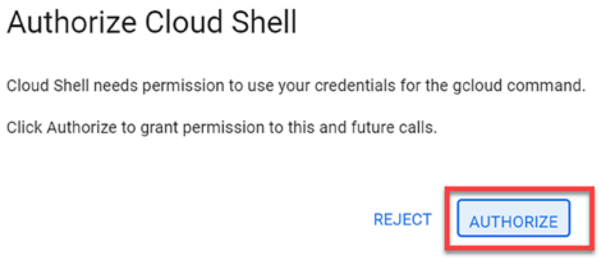
   The following screenshot shows the actions performed in *Steps 5* and *6*.



```
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = "3.5.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "3.5.1"
    }
  }
}

provider "google" {
  region = "us-central1" #update with your region
}

resource "random_string" "random" {
  length  = 4
  special = false
  upper   = false
}

resource "google_compute_network" "vpc_network" {
  name = "tf-demo-shell-${random_string.random.result}"
}
```

Figure 9.29: Writing Terraform configuration

7. Finally, inside the console, navigate inside this folder with the `cd terraform-demo` command and run the Terraform workflow by running the `init`, `plan`, and `apply` commands.
8. Note that during the first `apply` execution, the shell asks you to grant it permission to provision resources.



Authorize Cloud Shell

Cloud Shell needs permission to use your credentials for the gcloud command.

Click Authorize to grant permission to this and future calls.

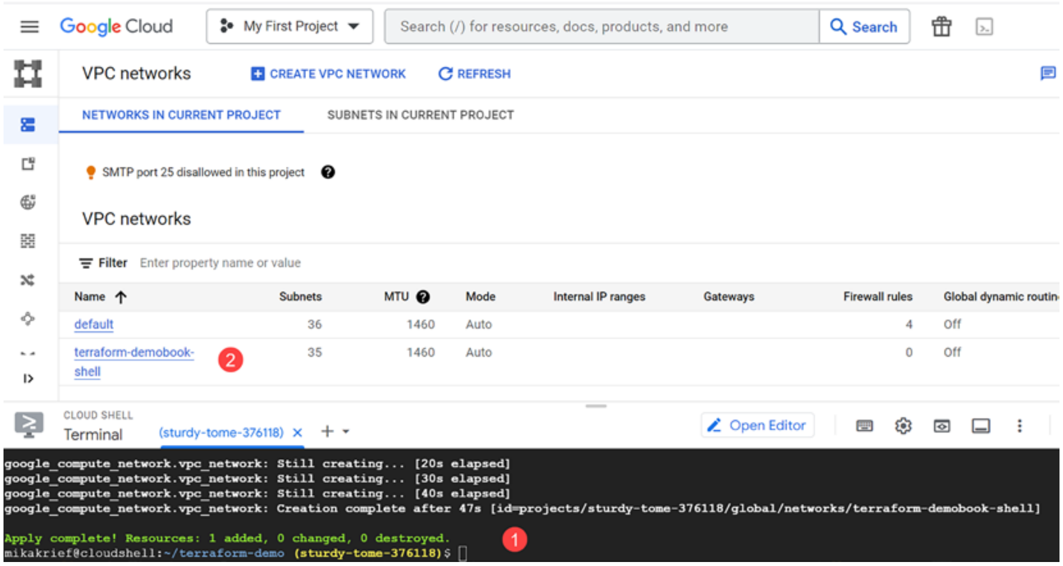
Figure 9.30: AWS sign-in button

At the end of the command execution, the resources are provisioned.

How it works...

In this recipe, we opened Cloud Shell in GCP to write a Terraform configuration using the Cloud Shell editor and run the Terraform commands inside the Cloud Shell console.

At the end of the execution of the `apply` command, we can see the provisioned resources. The following screenshot shows the resources created in *Step 7* with the `apply` command [1].


Google Cloud
My First Project
Search (/) for resources, docs, products, and more
Search
VPC networks
CREATE VPC NETWORK
REFRESH
NETWORKS IN CURRENT PROJECT
SUBNETS IN CURRENT PROJECT
SMTP port 25 disallowed in this project
VPC networks
Filter Enter property name or value
2
CLOUD SHELL
Terminal
(sturdy-tome-376118)
Open Editor
1

Name	Subnets	MTU	Mode	Internal IP ranges	Gateways	Firewall rules	Global dynamic routin
default	36	1460	Auto			4	Off
terraform-demobook-shell	35	1460	Auto			0	Off

```
google_compute_network.vpc_network: Still creating... [20s elapsed]
google_compute_network.vpc_network: Still creating... [30s elapsed]
google_compute_network.vpc_network: Still creating... [40s elapsed]
google_compute_network.vpc_network: Creation complete after 47s [id=projects/sturdy-tome-376118/global/networks/terraform-demobook-shell]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
mikakrief@cloudshell:~/terraform-demo (sturdy-tome-376118)$
```

Figure 9.31: GCP Cloud Shell terraform apply output and result in the console

We can see the output of the `apply` command and the VPC resource created [2].

There's more...

In this recipe, we use the cloud editor to write Terraform configurations. We can also use `git` commands to clone a Git repository to use an existing Terraform configuration.

For example, we can clone the GitHub repository of this book using the `git clone https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition.git` command and navigate to the `CHAP09/gcp-shell` folder.

If you use GCP Cloud Shell by default without any backend configuration, the state file will be stored in the Cloud Shell drive, so be careful to use a remote backend to store the Terraform state file.

See also

- The documentation of GCP Cloud Shell is available here: `https://cloud.google.com/shell/`
- A tutorial on the use of Terraform on GCP using Cloud Shell is available here: `https://github.com/hashicorp/terraform-getting-started-gcp-cloud-shell/blob/master/tutorial/cloudshell_tutorial.md`
- Getting started with GCP Cloud Shell: `https://github.com/hashicorp/terraform-getting-started-gcp-cloud-shell`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

10

Using Terraform for Docker and Kubernetes Deployment

When we talk about Terraform, the first thing that comes to mind is using Terraform for Infrastructure as Code to provision resources in the popular cloud providers, like Azure, AWS, or GCP.

Terraform can be used for more than just cloud provisioning. It has a multitude of providers that will allow you to automate any type of components or configuration that are not related to the cloud, such as files (which we studied in *Chapter 4*, *Using Terraform with External Data*), Docker, Kubernetes, and many other resources.

In the previous chapter, we learned about Terraform for Azure, AWS, and GCP. In this chapter, we will move on to another domain, which is the usage of Terraform in the context of Docker and Kubernetes. Indeed, with its simplicity, Terraform is becoming more and more of a tool for centralizing the automation of resource and component deployment using the same Infrastructure as Code language.

In this chapter, we will learn how to use Terraform to manipulate Docker images, and how to deploy applications in Kubernetes using the `kubernetes` provider and `helm` provider.

Then, we will learn how to apply GitOps practices with Terraform, with the usage of the Kubernetes Terraform controller.

In this chapter, we will cover the following recipes:

- Creating a Docker container using Terraform
- Deploying Kubernetes resources using Terraform

- Deploying a Helm chart in Kubernetes using Terraform
- Using a Kubernetes controller as a Terraform reconciliation loop

Technical requirements

To apply the recipes in this chapter, you need to have basic knowledge of Docker, Kubernetes, and Helm.

Below is a list of links to documentation for learning about these concepts and technologies:

- Docker: `https://docs.docker.com/`
- Kubernetes: `https://kubernetes.io/docs/home`
- Helm: `https://helm.sh/`

You can also read *Section 3* of the book *Learning DevOps: Second Edition* here `https://subscription.packtpub.com/book/cloud-and-networking/9781801818964/11#_ga=2.194054702.286366035.1676194628-504938521.1673192707`, specifically *Chapter 9*, *Containerizing Your Application with Docker*, and *Chapter 10*, *Managing Containers Effectively with Kubernetes*.

In the last two recipes, we will discuss more advanced concepts with GitOps and Kubernetes operators.

To apply the recipes of this chapter, we need to install these software requirements:

- Docker Engine: `https://docs.docker.com/engine/install/`
- kubectl: `https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/`
- The Helm CLI: `https://helm.sh/docs/intro/install/`

This chapter doesn't explain the provisioning of the cluster with Terraform, because that depends on your Kubernetes platform or cloud provider.

So, for all recipes of this chapter, we assume that we have an already-provisioned Kubernetes cluster.

If you want to provision a Kubernetes cluster in a cloud provider using Terraform, here are some HashiCorp tutorials:

- Provisioning an AKS cluster with Terraform: `https://developer.hashicorp.com/terraform/tutorials/kubernetes/aks`
- Provisioning an EKS cluster with Terraform: `https://developer.hashicorp.com/terraform/tutorials/kubernetes/eks`

- Provisioning a GKE cluster with Terraform: `https://developer.hashicorp.com/terraform/tutorials/kubernetes/gke`

Creating a Docker container using Terraform

To start this chapter dedicated to Docker and Kubernetes, we will learn how to automate Docker operations using Terraform.

Let's get started!

Getting ready

To complete this recipe, you will need to install Docker Engine by reading the documentation here: `https://docs.docker.com/engine/`. In this recipe, we assume that Docker is already installed and running on the same workstation on which we run Terraform.

In this recipe, we will discuss how to pull a Docker image named `mikaelkrief/demobook` from Docker Hub, and then run a container from this pulled image.

We will not learn how to build this Docker image from the dockerfile, but the source code for this Dockerfile is available here: `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP09/appdocker/Dockerfile`.

Now, in this recipe, we will learn how to pull this image and how to run it as a container using Terraform configuration.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP10/docker`.

How to do it…

To pull a Docker image and run it as a container, perform the following steps:

1. In a new file in `main.tf`, write the following Terraform configuration:

```
terraform {
  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
      version = "3.0.1"
    }
  }
}
```

2. In this file, add the following content:

```
provider "docker" {
}
```

3. Then, add the Terraform configuration:

```
resource "docker_image" "image" {
  name = "mikaelkrief/demobook:latest"
}
```

4. Continue to add the following content:

```
resource "docker_container" "container" {
  name   = "demo"
  image = docker_image.image.image_id
}
```

5. Finally, we run the basic Terraform workflow by executing the `terraform init`, `plan`, and `apply` commands.

How it works...

In *Step 1*, we create a new `main.tf` and we initialize it with the `terraform` block that specifies that we use the `kreuzwerker/docker` Terraform provider.

In *Step 2*, we add the configuration for the `docker` provider with empty configuration.

For more information about the `docker` provider configuration, read the documentation here: `https://registry.terraform.io/providers/kreuzwerker/docker/latest/docs`.

In *Step 3*, we add the Terraform resource `docker_image`, which pulls a Docker image; the `name` property indicates the name (name and tag of the image) to pull.

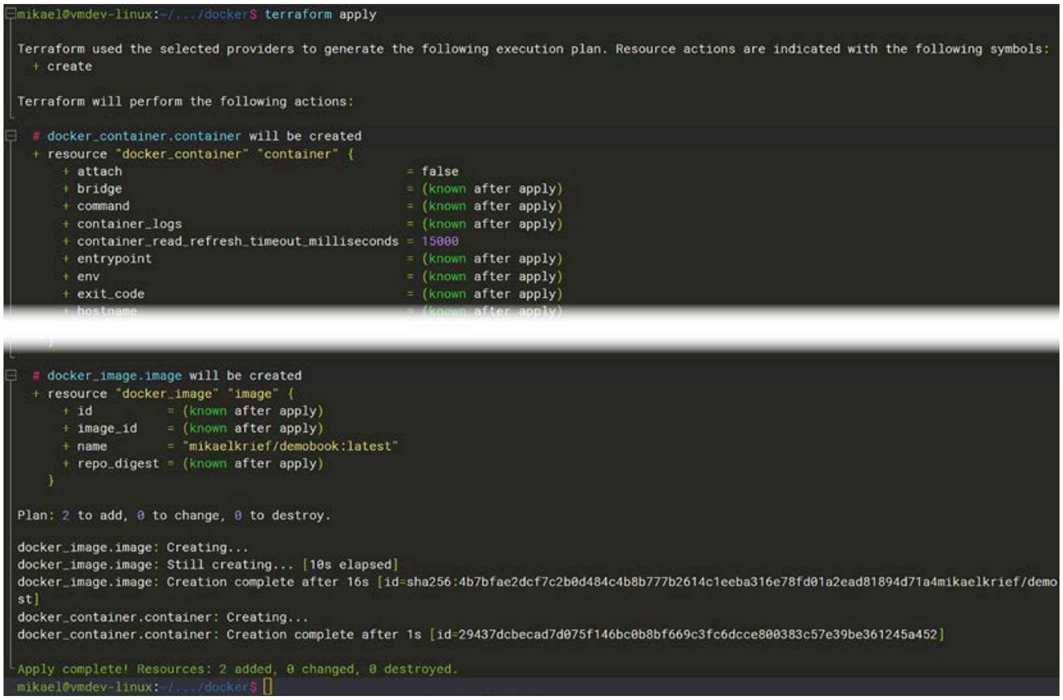
Here, in this recipe, we pull the image `mikaelkrief/demobook:latest`, which is published here: `https://hub.docker.com/repository/docker/mikaelkrief/demobook/general`.

Then, in *Step 4*, we add the resource `docker_container` to run a Docker container from the pulled image in *Step 3*.

For this resource, we indicate the container name "demo" and the image to run in the `image` property using the implicit Terraform dependency `docker_image.image.image_id` to use the image `Id` pulled from the above resource `docker_image` from *Step 3*.

Finally, to apply this Terraform configuration, we run the `terraform init`, `plan`, and `apply` commands.

The following image shows the execution of the `terraform apply` command on this configuration:



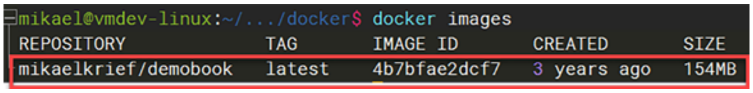
```
mikael@vmdev-linux:~/.../docker$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # docker_container.container will be created
  + resource "docker_container" "container" {
      + attach                                      = false
      + bridge                                      = (known after apply)
      + command                                     = (known after apply)
      + container_logs                              = (known after apply)
      + container_read_refresh_timeout_milliseconds = 15000
      + entrypoint                                  = (known after apply)
      + env                                         = (known after apply)
      + exit_code                                   = (known after apply)
      + hostname                                    = (known after apply)

  # docker_image.image will be created
  + resource "docker_image" "image" {
      + id          = (known after apply)
      + image_id    = (known after apply)
      + name        = "mikaelkrief/demobook:latest"
      + repo_digest = (known after apply)
    }

Plan: 2 to add, 0 to change, 0 to destroy.

docker_image.image: Creating...
docker_image.image: Still creating... [10s elapsed]
docker_image.image: Creation complete after 16s [id=sha256:4b7bfae2dcf7c2b0d484c4b8b777b2614c1eeba316e78fd01a2ead81894d71a4mikaelkrief/demo
st]
docker_container.container: Creating...
docker_container.container: Creation complete after 1s [id=29437dcbecad7d075f146bc0b8bf669c3fc6dcce800383c57e39be361245a452]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
mikael@vmdev-linux:~/.../docker$ 
```

Figure 10.1: Terraform apply command with docker provider

We can see that `terraform apply` command is working successfully.

Now, we can check the execution of this `terraform apply` above by running Docker commands. The first check is to verify that the image `mikaelkrief/demobook` is pulled into our host. For this, we run the Docker command `docker images`.

The output of this command is shown in the following image:



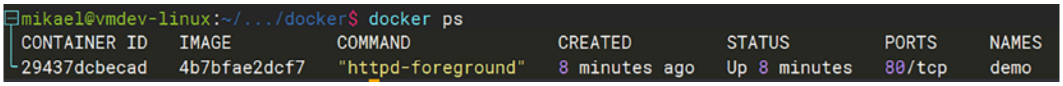
```
mikael@vmdev-linux:~/.../docker$ docker images
REPOSITORY             TAG       IMAGE ID       CREATED       SIZE
mikaelkrief/demobook   latest    4b7bfae2dcf7   3 years ago   154MB
```

Figure 10.02: List of Docker images pulled

We can see that the `mikaelkrief/demobook` image is pulled with **IMAGE ID** `4b7bfae2dcf7`.

The second check is to verify the list of containers running on this host; for this, we run the Docker command: `docker ps`.

The output of this command is shown in the following image:



```
mikael@vmdev-linux:~/.../docker$ docker ps
CONTAINER ID   IMAGE          COMMAND               CREATED         STATUS         PORTS     NAMES
29437dcbecad   4b7bfae2dcf7   "httpd-foreground"    8 minutes ago   Up 8 minutes   80/tcp    demo
```

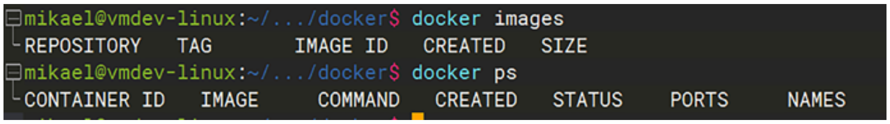
Figure 10.03: List of Docker containers

We can see that the container "`demo`" based on the Image ID `4b7bfae2dcf7` is running.

There's more...

Like in all Terraform workflows, we can delete the Docker container and the image by running the `terraform destroy` command.

The following image shows the output of the `docker images` and `docker ps` commands after the execution of the `terraform destroy` command:



```
mikael@vmdev-linux:~/.../docker$ docker images
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
mikael@vmdev-linux:~/.../docker$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Figure 10.04: Docker image and container deleted

The container and the image `mikaelkrief/demobook` have been deleted.

See also

- The documentation for the `kreuzwerker/docker` Terraform provider is available here: `https://registry.terraform.io/providers/kreuzwerker/docker/latest/docs`.
- Here is an article about Terraform and docker provider: `https://automateinfra.com/2021/03/29/how-to-build-docker-images-containers-and-docker-services-with-terraform-using-docker-provider/`.

Deploying Kubernetes resources using Terraform

There are several methods to deploy applications in Kubernetes. The default and basic method to deploy YAML specifications of Kubernetes resources is to use the Kubernetes CLI called `kubectl`. For more information, read the documentation here: `https://kubernetes.io/docs/reference/kubectl/`.

In some situations or use cases, when Terraform is well implemented in the company, instead of using different tools or CLIs, we want to automate the deployment of the cluster and the deployment of the applications using Terraform configuration.

In addition, the use of Terraform for Kubernetes resources deployment will allow you to see a preview of the changes when you apply the workflow.

In this recipe, we learn how to deploy Kubernetes resources (using object YAML specifications) in a Kubernetes cluster using Terraform.

Let's get started!

Getting ready

To complete this recipe, you will need to know about the Kubernetes resource specifications (the documentation is available here: `https://kubernetes.io/docs/concepts/`).

Before performing this recipe, you need to have a running Kubernetes cluster and the Kubernetes configuration file in `~/.kube/config`. For configuring and authenticating Kubernetes, set new environment variable called `KUBE_CONFIG_PATH` containing the path of the Kubernetes configuration file (for example, on Linux this can be "`~/.kube/config`").

In this recipe, in the same Terraform configuration, we will perform these operations:

- Create a new Kubernetes namespace
- In this namespace, deploy a Deployment resource and a Service resource

The deployed Pod will contain a container instance of this Docker image: `mikaelkrief/demobook`.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP10/app`.

How to do it…

To use Terraform to deploy a Kubernetes application, perform the following steps:

1. In a new file called `main.tf`, write the following Terraform configuration:

```
terraform {
  required_providers {
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = ~> 2.18"
    }
  }
}

provider "kubernetes" {
}
```

2. Then, add the following Terraform configuration:

```
resource "kubernetes_namespace" "ns" {
  metadata {
    labels = {
      mylabel = "demobook"
    }

    name = "myapp"
  }
}
```

3. Continue to add the following Terraform configuration:

```
resource "kubernetes_deployment" "deployment" {
  metadata {
    name      = "webapp"
    namespace = kubernetes_namespace.ns.metadata.0.name
    labels = {
      app = "webapp"
    }
  }

  spec {
    replicas = 2

    selector {
      match_labels = {
        app = "webapp"
      }
    }
```

```
      template {
        metadata {
          labels = {
            app = "webapp"
          }
        }

        spec {
          container {
            image = "mikaelkrief/demobook:latest"
            name  = "demobookk8s"
          }
        }
      }
    }
  }
```

4. Continue to add the following Terraform configuration:

```
resource "kubernetes_service" "service" {
  metadata {
    name      = "webapp"
    namespace = kubernetes_namespace.ns.metadata.0.name
  }
  spec {
    selector = {
      app = kubernetes_deployment.deployment.metadata.0.labels.app
    }

    port {
      port        = 80
      target_port = 80
      node_port   = 31001
    }

    type = "NodePort"
  }
}
```

5. Finally, in this folder, run the basic Terraform workflow by executing the `terraform init`, `plan`, and `apply` commands.

How it works...

In *Step 1*, we create a new `main.tf` file and we add the `terraform` block, in which we specify the source and the version of the `kubernetes` Terraform provider (see the documentation here: `https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs`).

We also configure the `kubernetes` provider to use the Kubernetes configuration file generated in the requirements of this recipe.

In *Step 2*, we add the namespace configuration using the `kubernetes_namespace` resource by specifying the metadata `labels` and `name` for the namespace.

In *Step 3*, we add the Kubernetes `deployment` object configuration using the `kubernetes_deployment` resource by specifying:

- The name of the deployment: `webapp`
- The namespace where the deployment will be stored: using the dependencies `kubernetes_namespace.ns.metadata.0.name`
- The replica number : `2`
- The container image: `mikaelkrief/demobook:latest`

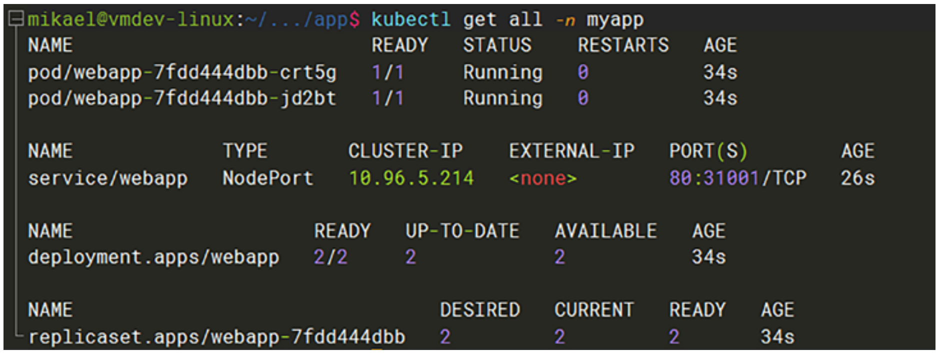
In *Step 4*, we add the Kubernetes `service` object using the `Kubernetes_service` resource by specifying the `port` exposition and the type of the service as `NodePort`.

Finally, in *Step 5*, we deploy this application running the Terraform workflow by executing the `terraform init`, `plan`, and `apply` commands.

At the end of the `apply` execution command, we run the following `kubectl` command to check that all resources are deployed successfully:

```
kubectl get all -n myapp
```

The following image shows the output of this command:



```
mikael@vmdev-linux:~/.../app$ kubectl get all -n myapp
NAME                           READY   STATUS    RESTARTS   AGE
pod/webapp-7fdd444dbb-crt5g   1/1     Running   0          34s
pod/webapp-7fdd444dbb-jd2bt   1/1     Running   0          34s

NAME             TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
service/webapp   NodePort   10.96.5.214   <none>        80:31001/TCP   26s

NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/webapp   2/2     2            2           34s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/webapp-7fdd444dbb   2         2         2       34s
```

Figure 10.05: Get a list of Pods in the myapp namespace

We can see in the above output:

- The 2 Pod replicas
- The service of type `NodePort`
- The deployment Kubernetes resource

There's more…

In this recipe, in the sample application, we use classic Kubernetes resources like a namespace, Deployment, and Service, and to deploy them, we use the Kubernetes provider.

To learn with more examples of **Custom Resource Deployment** (**CRD**) using Terraform, read the tutorial here: `https://developer.hashicorp.com/terraform/tutorials/kubernetes/kubernetes-crd-faas`.

Moreover, for your existing YAML Kubernetes specifications, if you want to migrate them into a Terraform configuration in **HashiCorp Configuration Language** (**HCL**), you can use the `k2tf` tool, available here: `https://github.com/sl1pm4t/k2tf`.

Another point – to delete all the deployed resources in Kubernetes, run the command `terraform destroy`.

See also

- The documentation for the Terraform Kubernetes provider is available here: `https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs`.

Deploying a Helm chart in Kubernetes using Terraform

In the previous recipe, we learned how to deploy Kubernetes applications resources using Terraform and the `kubernetes` provider.

Among the different ways to deploy an application in Kubernetes, there is also the possibility to use (or deploy) a Helm chart.

We will not explain here in detail the use and creation of a Helm chart, but to explain simply, a Helm chart is a package that contains a template of Kubernetes resources to deploy.

For all the details about Helm chart usage and creation, read the documentation here: `https://helm.sh/docs/`.

In this recipe, we will learn how to deploy a Helm chart in Kubernetes using Terraform.

Let's get started!

Getting ready

For the cluster provisioning, we will use the existing Kubernetes cluster.

To illustrate this recipe, we will deploy a sample application, which is the nginx ingress controller. The documentation on the nginx ingress controller is available here: `https://artifacthub.io/packages/helm/ingress-nginx/ingress-nginx` and here `https://kubernetes.github.io/ingress-nginx`.

Note that in this recipe, we will deploy `ingress-nginx`, which is just a sample application; all the steps can be applied to all and any Helm chart packages.

To run a Helm command to check the deployment, we need to install the Helm CLI by following the documentation here: `https://helm.sh/docs/intro/install/`.

Before performing this recipe, you need to have a running Kubernetes cluster and the Kubernetes configuration file in `~/.kube/config`. For configuring and authenticating Kubernetes, set new environment variable called `KUBE_CONFIG_PATH` containing the path of the Kubernetes configuration file (for example, on Linux this can be "`~/.kube/config`").

The source code for this chapter is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP10/helm`.

How to do it...

To deploy a Helm chart using Terraform, perform the following steps:

1. In a new file `main.tf`, write the following Terraform configuration:

```
terraform {
  required_providers {
    helm = {
      source  = "hashicorp/helm"
      version = "~> 2.7"
    }
  }
}

provider "helm" {
  kubernetes {
    config_path = pathexpand("~/.kube/config")
  }
}
```

2. Then, add the following Terraform configuration:

```
resource "helm_release" "nginx_ingress" {
  name             = "ingress"
  repository       = "https://kubernetes.github.io/ingress-nginx"
  chart            = "ingress-nginx"
  version          = "4.5.2"
  namespace        = "ingress"
  create_namespace = true
  wait             = true

```

```
    set {
      name  = "controller.replicaCount"
      value = 2
    }

    set {
      name  = "controller.service.type"
      value = "NodePort"
    }
  }
```

3. Finally, run the basic Terraform workflow by running the `terraform init`, `plan`, and `apply` commands.

How it works…

In *Step 1*, we create a new `main.tf` file and we add the `terraform` block in which we specify the source and the version of the `helm` Terraform provider (see the documentation here: `https://registry.terraform.io/providers/hashicorp/helm/latest`).

We also configure the `helm` provider to use the Kubernetes configuration file generated in the *Deploying Kubernetes resources using Terraform* recipe of this chapter.

In *Step 2*, in this same file, we add the Terraform resource `helm_release` to configure the property of this Helm release.

Inside this resource configuration, we add a `set` block to override the default values of `values.yaml` provided in the Helm chart.

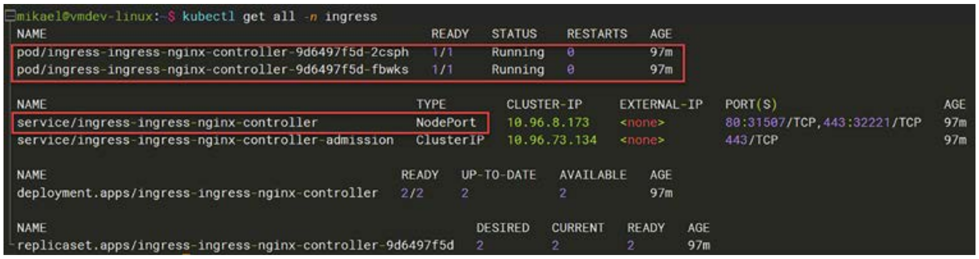
Here, we specify the version of the Helm chart to use, that we want 2 Pod replicas, and the service type to be `NodePort`.

Another important configuration is `create_namespace = true` to create the namespace `ingress` directly using Helm.

In *Step 3*, we deploy this Helm chart by running the Terraform workflow, which is itself run by the `terraform init`, `plan`, and `apply` commands.

At the end of the `apply` command, we check the deployment of the Kubernetes resources and then we check the deployment of the Helm release by running the following commands:

1. Run the command `kubectl get all -n ingress` to check the status of all resources deployed in the `ingress` namespace. The following image shows the output of this command:

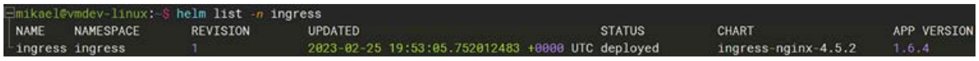


```
mikael@vmdev-linux:~$ kubectl get all -n ingress
NAME                                                READY   STATUS    RESTARTS   AGE
pod/ingress-ingress-nginx-controller-9d6497f5d-2csph   1/1     Running   0          97m
pod/ingress-ingress-nginx-controller-9d6497f5d-fbwks   1/1     Running   0          97m

NAME                                         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)                      AGE
service/ingress-ingress-nginx-controller     NodePort    10.96.8.173   <none>        80:31507/TCP,443:32221/TCP   97m
service/ingress-ingress-nginx-controller-admission   ClusterIP   10.96.73.134   <none>   443/TCP                  97m

NAME                                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ingress-ingress-nginx-controller   2/2     2            2           97m

NAME                                                         DESIRED   CURRENT   READY   AGE
replicaset.apps/ingress-ingress-nginx-controller-9d6497f5d   2         2         2       97m
```

Figure 10.6: Check all resources in the ingress namespace

   We can see that all resources are ready, the 2 Pod replicas, and that the service type of the controller is `NodePort`.

2. Run the command `helm list -n ingress` to check the deployment of the Helm release. The following image shows the output of this command:



```
mikael@vmdev-linux:~$ helm list -n ingress
NAME     NAMESPACE   REVISION   UPDATED                                   STATUS     CHART                 APP VERSION
ingress  ingress     1          2023-02-25 19:53:05.752012483 +0000 UTC   deployed   ingress-nginx-4.5.2   1.6.4
```

Figure 10.7: Get Helm list for the ingress namespace

We can see the ingress release and the deployment status is **deployed**.

There's more…

Here in this recipe, we can see a basic usage of the `helm_release` Terraform resource by setting values (replica number and service type) of the Helm chart directly in the resource configuration, we can also override the `values` property using `values.yaml` . For more information, read the `helm_release` documentation here: `https://registry.terraform.io/providers/hashicorp/helm/latest/docs/resources/release#values`.

Another consideration is if we want to have more control over the Kubernetes namespace to add labels, we can also create it using the Kubernetes provider, and reference it inside the `helm_resource` resource.

The following configuration shows sample code for the use of the `kubernetes` and `helm` providers together:

```
resource "kubernetes_namespace" "ns" {
  metadata {
    labels = {
      mylabel = "ingress"
    }
    name = "ingress"
  }
}

resource "helm_release" "nginx_ingress" {
  name             = "ingress"
  repository       = "https://kubernetes.github.io/ingress-nginx"
  chart            = "ingress-nginx"
  version          = "4.5.2"
  namespace        = kubernetes_namespace.ns.metadata.0.name
  create_namespace = true
  wait             = true
…
}
```

In the above configuration, we use implicit dependency to link the namespace at the release.

See also

- The `helm` provider documentation is available here: `https://registry.terraform.io/providers/hashicorp/helm/latest/docs`.

Using a Kubernetes controller as a Terraform reconciliation loop

In the previous chapter's recipes, we learned how to use Terraform to deploy a Docker container, provision a Kubernetes cluster, and deploy applications in Kubernetes using different tools.

In this recipe, we will learn how to perform a **reconciliation loop** to provision infrastructure using Terraform and reapply the Terraform workflow at each code change.

By using Terraform reconciliation, you can:

- Ensure that the state of your infrastructure always matches the desired state defined in the Terraform code.
- Automate the deployment of infrastructure resources based on changes to the Git repository, reducing the risk of errors and improving the speed of deployments.

So, we will learn in this recipe how to apply infrastructure changes coded with Terraform using a Kubernetes controller in Kubernetes.

Let's get started!

Getting ready

To complete this recipe, we need some prerequisites:

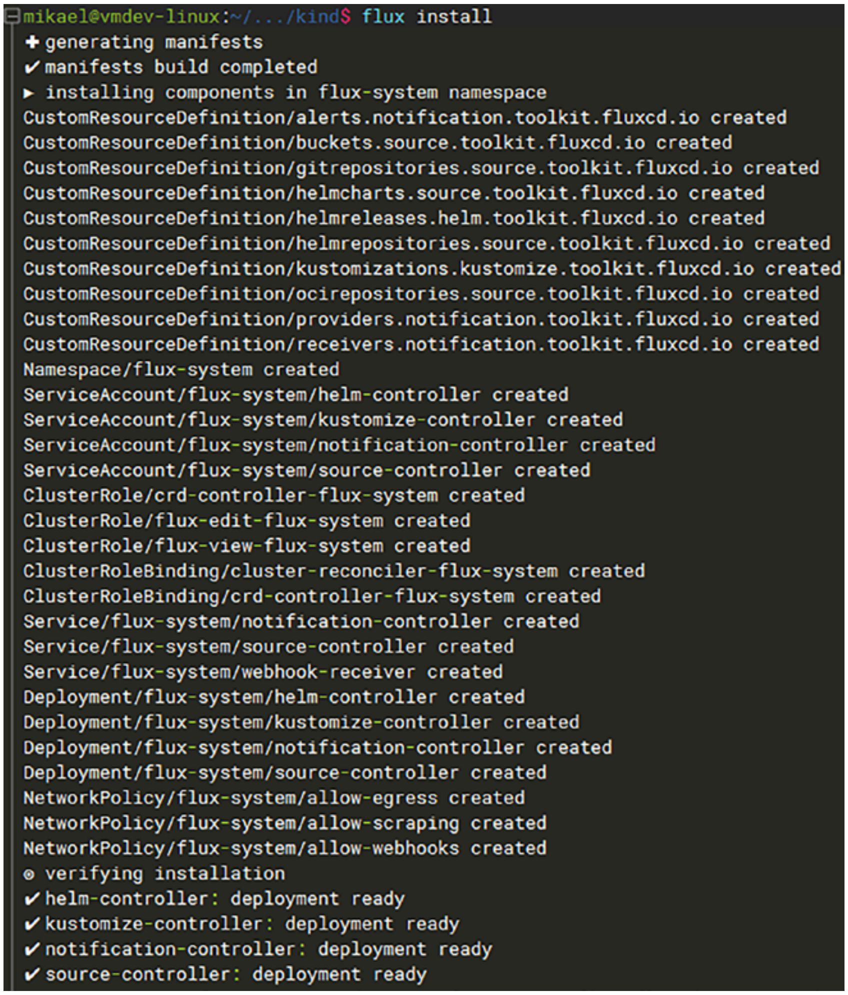
- Store the Terraform configuration in a Git repository – for this, we will use the existing Terraform configuration that provisioned the Azure Web Apps app, detailed in the recipe, *Building Azure serverless infrastructure with Terraform* in *Chapter 8, Provisioning Azure Infrastructure with Terraform*, and the source code is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/webapp`.
- Have an existing Kubernetes cluster.
- Install Helm CLI, to install Flux components (the commands will be detailed in the recipe).
- Add a `KUBECONFIG` environment variable containing the path of the Kubernetes configuration file, for example `~/.kube/config`.

There are many tools to provide GitOps on Kubernetes; for this recipe, we will use Weave Flux, and for Terraform, we will use Flux `tf-controller`. We will see their installation and their use in the body of the recipe.

As a requirement, we will install the **Flux** CLI by referring to the documentation here: `https://fluxcd.io/flux/installation/`, depending on your operating system. Then, we will install Flux operator resources in our Kubernetes cluster by running the command `flux install`.

This Flux command creates the `flux-system` namespace and all required resources in our cluster.

The following image shows the output of this command:



```
mikael@vmdev-linux:~/.../kind$ flux install
✚ generating manifests
✔ manifests build completed
► installing components in flux-system namespace
CustomResourceDefinition/alerts.notification.toolkit.fluxcd.io created
CustomResourceDefinition/buckets.source.toolkit.fluxcd.io created
CustomResourceDefinition/gitrepositories.source.toolkit.fluxcd.io created
CustomResourceDefinition/helmcharts.source.toolkit.fluxcd.io created
CustomResourceDefinition/helmreleases.helm.toolkit.fluxcd.io created
CustomResourceDefinition/helmrepositories.source.toolkit.fluxcd.io created
CustomResourceDefinition/kustomizations.kustomize.toolkit.fluxcd.io created
CustomResourceDefinition/ocirepositories.source.toolkit.fluxcd.io created
CustomResourceDefinition/providers.notification.toolkit.fluxcd.io created
CustomResourceDefinition/receivers.notification.toolkit.fluxcd.io created
Namespace/flux-system created
ServiceAccount/flux-system/helm-controller created
ServiceAccount/flux-system/kustomize-controller created
ServiceAccount/flux-system/notification-controller created
ServiceAccount/flux-system/source-controller created
ClusterRole/crd-controller-flux-system created
ClusterRole/flux-edit-flux-system created
ClusterRole/flux-view-flux-system created
ClusterRoleBinding/cluster-reconciler-flux-system created
ClusterRoleBinding/crd-controller-flux-system created
Service/flux-system/notification-controller created
Service/flux-system/source-controller created
Service/flux-system/webhook-receiver created
Deployment/flux-system/helm-controller created
Deployment/flux-system/kustomize-controller created
Deployment/flux-system/notification-controller created
Deployment/flux-system/source-controller created
NetworkPolicy/flux-system/allow-egress created
NetworkPolicy/flux-system/allow-scraping created
NetworkPolicy/flux-system/allow-webhooks created
◎ verifying installation
✔ helm-controller: deployment ready
✔ kustomize-controller: deployment ready
✔ notification-controller: deployment ready
✔ source-controller: deployment ready
```

Figure 10.8: Installation of Flux in Kubernetes

All the Flux components are installed on the Kubernetes cluster in the `flux-system` namespace.

In this recipe, we will learn from the beginning how to apply Terraform configuration using Flux's `tf-controller` in Kubernetes.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP10/gitops`.

How to do it...

Perform the following steps to apply Terraform configuration with `tf-controller`:

1. Install Flux `tf-controller` with Helm by running the two following commands:

```
helm repo add tf-controller https://weaveworks.github.io/tf-
controller/

helm upgrade -i tf-controller tf-controller/tf-controller
--namespace flux-system
```

2. Then, create a new file called `tf-controller-azurerm.yaml` and add the following code inside it:

```
---
apiVersion: source.toolkit.fluxcd.io/v1beta1
kind: GitRepository
metadata:
  name: azurerm-demo
  namespace: flux-system
spec:
  interval: 30s
  url: https://github.com/PacktPublishing/Terraform-Cookbook-Second-
Edition
  ref:
    branch: main
```

3. Append the content of this YAML file by adding the following content:

```
---
apiVersion: infra.contrib.fluxcd.io/v1alpha1
kind: Terraform
metadata:
  name: azurerm-demo
  namespace: flux-system
spec:
  path: ./CHAP08/sample-app/
  interval: 1m
  approvePlan: auto
  sourceRef:
    kind: GitRepository
    name: azurerm-demo
    namespace: flux-system
  runnerPodTemplate:
    spec:
      env:
      - name: ARM_CLIENT_ID
        value: "<your azure client id>"
      - name: ARM_TENANT_ID
        value: "<your azure tenant id>"
      - name: ARM_CLIENT_SECRET
        value: "<your azure client secret>"
      - name: ARM_SUBSCRIPTION_ID
        value: "<your azure subscription id>"
```

4. Finally, deploy this Kubernetes resource by running the following `kubectl` command:

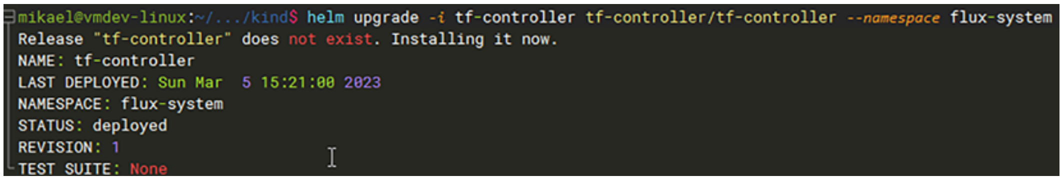
```
kubectl apply -f tf-controller-azurerm.yaml
```

Let's look at all these steps in detail.

How it works...

In *Step 1*, we add the `tf-controller` registry locally and then we install the Terraform controller called `tf-controller` using the `helm upgrade -install` command from the `tf-controller` Helm registry inside the `flux-system` namespace.

The following image shows the output of this `helm` command:



```
mikael@vmdev-linux:~/.../kind$ helm upgrade -i tf-controller tf-controller/tf-controller --namespace flux-system
Release "tf-controller" does not exist. Installing it now.
NAME: tf-controller
LAST DEPLOYED: Sun Mar  5 15:21:00 2023
NAMESPACE: flux-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Figure 10.9: Install tf-control using the Helm chart

The Helm chart is installed successfully.

To check the installation status of the Helm chart, run the command `helm list -n flux-system`, The output status value must be `deployed`.

> Note that in this recipe, we install the `tf-controller` using Helm, but we can also use the `kubectl` command, as explained in this documentation: `https://weaveworks.github.io/tf-controller/getting_started/#installation`.

By this point, we have installed all the necessary components to run our Terraform configuration inside the `tf-controller`.

Then, in *Step 2* and *Step 3*, we create a new file called `tf-controller-azure.yaml`, which will contain all the configurations to run the Terraform configuration.

In the first part of this file, we add the kind resource `GitRepository` in which we configure the URL and the branch name of the Git repository where the Terraform configuration is stored. (Here, in this recipe, the configured repository is `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition` and the controller will target only the `main` branch of this repository).

In the second part of this file, we add the `Terraform` kind Kubernetes resource, in which we configure:

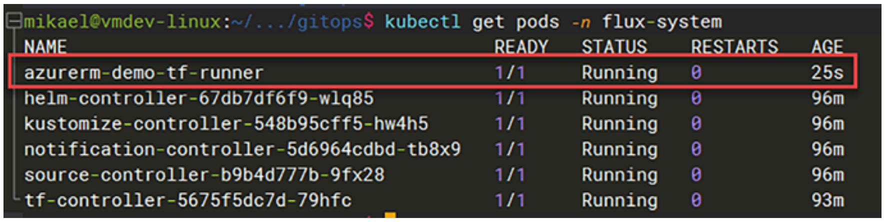
- The `path` where the Terraform configuration is inside the Git repository (here, our configuration to apply is in the `CHAP08/sample-app` folder).
- The `autoapprove` option to indicate to the controller to perform the complete Terraform workflow with `init`, `plan`, and `apply` without the user performing any operation.
- The `name` of the `GitRepository` resource added just before this file.
- The four Azure environment variables to authenticate the `azurerm` provider; for more information, read the *Protecting the Azure credential provider* recipe in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*.

Finally, in *Step 4*, we deploy these two new Kubernetes configurations (the `GitRepository` and `Terraform` kinds) in our Kubernetes cluster by running the `kubectl apply -f <file>` command.

Now, what is interesting is to understand what happened once this command was executed.

When these two new Kubernetes resources are deployed, the `tf-controller` will create a new Pod that will run the Terraform workflow on the specified configuration in the Git repository.

The following image shows the new Pod called `azurerm-demo-tf-runner` in the `flux-system` namespace:



```
mikael@vmdev-linux:~/.../gitops$ kubectl get pods -n flux-system
NAME                                       READY   STATUS    RESTARTS   AGE
azurerm-demo-tf-runner                     1/1     Running   0          25s
helm-controller-67db7df6f9-wlq85           1/1     Running   0          96m
kustomize-controller-548b95cff5-hw4h5      1/1     Running   0          96m
notification-controller-5d6964cdbd-tb8x9   1/1     Running   0          96m
source-controller-b9b4d777b-9fx28          1/1     Running   0          96m
tf-controller-5675f5dc7d-79hfc             1/1     Running   0          93m
```

Figure 10.10: Get Pods in the flux-system namespace

And once the execution of Terraform is finished, it automatically destroys itself and the Azure resources are provisioned.

Moreover, any commit made on the main branch will re-trigger the execution of this Pod, which will re-execute the Terraform workflow on the changes made to the infrastructure configuration, and thus ensure that the infrastructure (in this case, Azure) will always be up to date with respect to its Terraform configuration.

There's more...

In this recipe, we see how to apply GitOps for Terraform using Weave `tf-controller`; we can also use **ArgoCD** tools in tandem; for more information, read the documentation here: `https://www.cncf.io/blog/2022/09/30/how-to-gitops-your-terraform/`.

Another Kubernetes operator allows you to apply GitOps practices for Terraform to Kubernetes by using the **Terraform Operator**, and for more information, refer to the documentation here: `http://tf.isaaguilar.com/`.

Additionally, in *Step 1*, we use the Helm command line to install the `tf-controller` Helm chart; we can instead use the Terraform Helm provider (for more details, read the *Deploying a Helm chart in Kubernetes using Terraform* recipe earlier in this chapter) using the following Terraform configuration:

```
resource "helm_release" "tf-controller" {
  name             = "tf-controller"
  repository       = "https://weaveworks.github.io/tf-controller/"
  chart            = "tf-controller"
  namespace        = "flux-system"
  wait             = true
}
```

The complete source code for this Terraform configuration is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP10/tf-controller`.

Finally, in this recipe, we see how to automatically deploy Terraform configuration using a Kubernetes controller; there are two other practices or tools to deploy Terraform configuration.

The first method is to use a CI/CD pipeline in pipeline tools like GitHub Actions, Jenkins, Azure DevOps, and more. We will see this in detail in *Chapter 13*, *Automating Terraform Execution in a CI/CD Pipeline*.

The second method is to use Terraform Cloud, which contains automated pipelines. We will see this in detail in *Chapter 14*, *Using Terraform Cloud to Improve Team Collaboration*.

See also

- The Weave `tf-controller` documentation is available here: `https://weaveworks.github.io/tf-controller/`
- The GitHub source code of the `tf-controller` is available here: `https://github.com/weaveworks/tf-controller`
- Article from the CNCF on using the `tf-controller`: `https://www.cncf.io/blog/2022/09/30/how-to-gitops-your-terraform/`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

11

Running Test and Compliance Security on Terraform Configuration

When writing Terraform configuration, it is very important to integrate a test phase into the Terraform workflow.

Indeed, when provisioning an infrastructure using a Terraform configuration, one must be careful not to bypass security rules, not to introduce vulnerabilities (even unintentionally), and to respect the company's conventions.

For these reasons, it is necessary to introduce one or more test phases that will be applied to our Terraform configuration.

Exactly as for an application, there are several types of tests (for more information, see the explanation of the test pyramid here: `https://martinfowler.com/articles/practical-test-pyramid.html`) for a Terraform configuration.

For a Terraform configuration, here are the different types of tests:

- The unit test: This aims to test the Terraform configuration at the lowest level, by checking bits of resources without dependency on external elements. The unit tests are performed before the execution of the `terraform apply` command, i.e., on the HCL code or the result of the `terraform plan`. Unit tests can consist of checking the syntax of the code and the consistency of the variables by executing the `terraform validate` command, which we saw in detail in *Chapter 6*, *Applying a Basic Terraform Workflow*, in the *Validating the code syntax recipe*, or analyzing the result of `terraform plan` and checking if the result obtained is the expected one.

- Plan analysis for compliance: This analysis of the plan result can be done either by using third-party tools, which have built-in rules like **tflint**, **checkov**, or **tfsec**, as we will see in this chapter, or by using a framework, and in this case, we will write our own rules, as we will see in this chapter, with PowerShell or Python. This analysis is performed just after the `terraform plan` execution, so if the analysis fails, `terraform apply` is not executed.
- The contact tests: These test the correct input of variables by using a condition expression on the variables that we quickly learned about in *Chapter 2, Writing Terraform Configurations* in the *Manipulating variables recipe* – for more details, read the documentation here: `https://developer.hashicorp.com/terraform/language/expressions/custom-conditions`.
- The integration tests: These test the correct provisioning of resources. These tests are most often used on Terraform modules to test that a Terraform configuration using this Terraform module gets the expected input. The workflow of the integration test is to provision the resources using the Terraform configuration; run some tests, like for connectivity or security; and, at the end, destroy all provisioned resources. This workflow can be operated using development scripts like PowerShell, Python, Bash, Ruby, and Golang or using a testing framework like **Terratest** (which we will learn about in the *Testing Terraform module code with Terratest recipe* in this chapter) or **Kitchen-Terraform** (which we will learn about in the recipe *Testing the Terraform configuration using Kitchen-Terraform* of this chapter).

For more details on testing Terraform, read the HashiCorp blog here: `https://www.hashicorp.com/blog/testing-hashicorp-terraform`.

In the first part of this chapter, we will learn how to use several methods and tools to test Terraform configuration by writing custom rules and policies using PowerShell or Python. We will also learn about the best tests to perform on Terraform configurations using tools like `terraform-compliance`, `tfsec`, and **Open Policy Agent** (**OPA**).

Then, in the second part of this chapter, we will learn how to perform an integration test on Terraform modules using Terratest or `Kitchen-Terraform`.

Finally, in the last part of the chapter, we will learn to use the new integrated Terraform module integration test to write a test directly in the HCL Terraform configuration.

Note that in all recipes of this chapter, we intentionally will start with Terraform configurations that are not compliant to show the tests error results; then, we will fix the Terraform configuration to be compliant with the tests and we will re-run the tests to show the result of the successful test execution.

In this chapter, we will cover the following recipes:

- Using PowerShell's Pester framework to perform Terraform testing
- Testing the Terraform configuration using Python
- Using OPA to check the Terraform configuration
- Using `tfsec` to analyze the compliance of the Terraform configuration
- Applying Terraform compliance with `terraform-compliance`
- Testing Terraform module code with Terratest
- Testing the Terraform configuration using `Kitchen-Terraform`
- Using the new integrated Terraform module integration test

Technical requirements

To complete the recipes in this chapter, we need to install some software requirements:

- Python – the installation documentation is available here: `https://www.python.org/downloads/`
- PowerShell or PowerShell Core – the installation documentation is available here: `https://learn.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-7.3`
- Golang – the installation documentation is available here: `https://go.dev/`
- Ruby – the installation documentation is available here: `https://www.ruby-lang.org/en/documentation/installation/`

The source code for this chapter is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11`.

Using PowerShell's Pester framework to perform Terraform testing

In this recipe, we will learn how to write and execute tests on our Terraform configuration using a testing framework.

This framework called **Pester** is a PowerShell library that allows us to write several types of tests.

With Pester, you can write tests that verify the behavior of your PowerShell scripts and ensure that they meet the expected requirements. Pester supports a wide range of tests, including unit tests, integration tests, and acceptance tests. It also includes powerful mocking capabilities, which can help you simulate complex scenarios and test your scripts in isolation.

Pester uses a simple syntax that is easy to understand and learn. Tests are written in PowerShell script files, which makes it easy to integrate them into your development workflow. Pester can be run from the command line, which makes it easy to automate your testing process and integrate it into your CI/CD pipeline.

In addition to testing PowerShell scripts, Pester can also be used to test other types of code, such as JSON or XML files. It is also extensible, which means you can write custom extensions and add-ons to support your specific testing needs.

Overall, Pester is a powerful and flexible testing framework that can help you improve the quality and reliability of your PowerShell scripts and other code.

So, we will learn how to write tests in PowerShell using Pester and then we will see their execution.

With Pester, tests are executed just after the `terraform plan` command, and before `terraform apply`, so that if the tests are failing, `terraform apply` will not continue.

Let's get started!

Getting ready

To complete this recipe, you will need to know about PowerShell scripting.

The software requirements are to install these in order:

1. PowerShell or PowerShell Core – refer to the documentation here: `https://learn.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-7.3`.
2. Then, install the PowerShell module Pester, the documentation for which is available here: `https://pester.dev/docs/introduction/installation`.

Like most of the book's recipes, this recipe can be run on Windows, Linux, or macOS.

The goal of this recipe is to provide a basic example of how to write and execute tests in Pester, which tests to make sure that the Azure storage account name is compliant and that the only access to the account is through HTTPS.

In this recipe, we will not detail the steps for writing the Terraform configuration that provisions the Azure storage account (which is detailed in the official documentation here: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/storage_account`). The code source for this Terraform configuration is the following Terraform configuration:

```
resource "azurerm_storage_account" "storage" {
  name                     = "sademotestpester1"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = false
}
```

For the demo of the recipe, in this above Terraform configuration, we explicitly configure the storage to be accessible via HTTP and name it `sademotestpester1`.

The complete source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/tf-pester`.

How to do it…

To write Pester tests, perform the following steps:

1. Inside the folder that contains our Terraform configuration, create a new PowerShell file called `storage.test.ps1`.
2. Start to write in this new file by adding the following `Describe` block:

   ```
   Describe "Azure Storage Account Compliance" {
   ….
   }
   ```

3. In this `Describe` block, add the following code:

   ```
       BeforeAll -ErrorAction Stop {
           Write-Host 'Performing terraform init...'
           terraform init
           Write-Host 'Performing terraform validate...'
           terraform validate
           Write-Host 'Performing terraform plan...'
           terraform plan -out terraform.plan
           Write-Host 'Converting the plan output to JSON...'
           $plan = terraform show -json terraform.plan  | ConvertFrom-
   Json
       }
   ```

4. Just after this above code, add the following code (inside the `Describe` block):

```
    It "should have the correct name" {
        $expectedStorageAccountName = "sademotestpester123"
        $storageAccountobj = $plan.resource_changes  | Where-Object
{ $_.address -eq 'azurerm_storage_account.storage' }
        $saName = $storageAccountobj.change.after.name
        $saName | Should -Be $expectedStorageAccountName
    }
```

5. Continue to add the following code (inside the `Describe` block):

```
    It "should be access only with HTTPS" {
        $expectedsaAccessHttps = "true"
        $storageAccountobj = $plan.resource_changes  | Where-Object
{ $_.address -eq 'azurerm_storage_account.storage' }
        $saAccessHttps = $storageAccountobj.change.after.enable_
https_traffic_only
        $saAccessHttps | Should -Be $expectedsaAccessHttps
    }
```

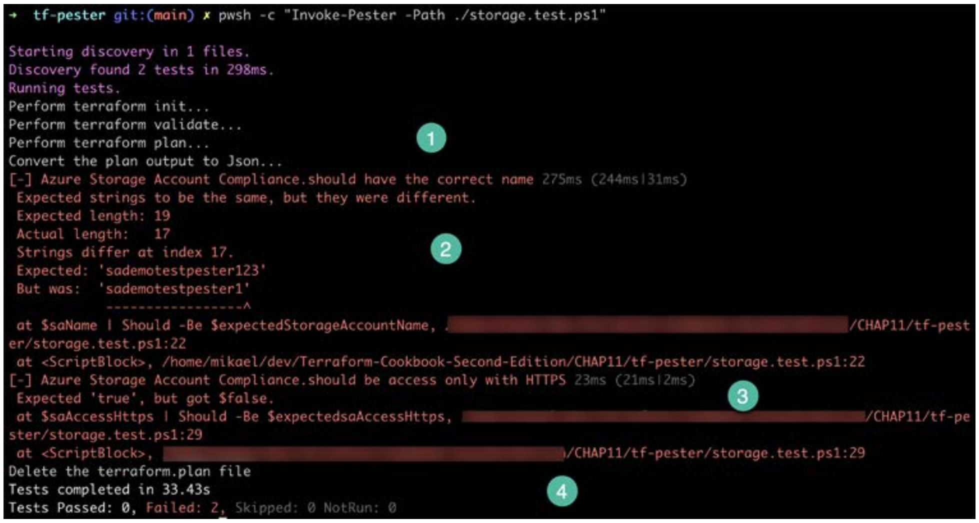
6. Finish writing the test by adding the following code (inside the `Describe` block):

```
    AfterAll {Write-Host 'Delete the terraform.plan file'
        Remove-Item -Path .\terraform.plan
    }
```

7. Finally, run Pester tests by executing the following command inside the folder that contains the PowerShell Pester script:

```
pwsh -c "Invoke-Pester -Path ./storage.test.ps1"
```

The following image shows the output of this command (using as a target the initial Terraform configuration that explicitly was not compliant):



```
→ tf-pester git:(main) ✗ pwsh -c "Invoke-Pester -Path ./storage.test.ps1"

Starting discovery in 1 files.
Discovery found 2 tests in 298ms.
Running tests.
Perform terraform init...
Perform terraform validate...
Perform terraform plan...
Convert the plan output to Json...
[-] Azure Storage Account Compliance.should have the correct name 275ms (244ms|31ms)
 Expected strings to be the same, but they were different.
 Expected length: 19
 Actual length:   17
 Strings differ at index 17.
 Expected: 'sademotestpester123'
 But was:  'sademotestpester1'
           -----------------^
 at $saName | Should -Be $expectedStorageAccountName, [redacted]/CHAP11/tf-pest
er/storage.test.ps1:22
 at <ScriptBlock>, /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/tf-pester/storage.test.ps1:22
[-] Azure Storage Account Compliance.should be access only with HTTPS 23ms (21ms|2ms)
 Expected 'true', but got $false.
 at $saAccessHttps | Should -Be $expectedsaAccessHttps, [redacted]/CHAP11/tf-pe
ster/storage.test.ps1:29
 at <ScriptBlock>, [redacted]/CHAP11/tf-pester/storage.test.ps1:29
Delete the terraform.plan file
Tests completed in 33.43s
Tests Passed: 0, Failed: 2, Skipped: 0 NotRun: 0
```

Figure 11.01: Terraform test with Pester execution with error

In this output, we can see the 4 operations of the tests as follows:

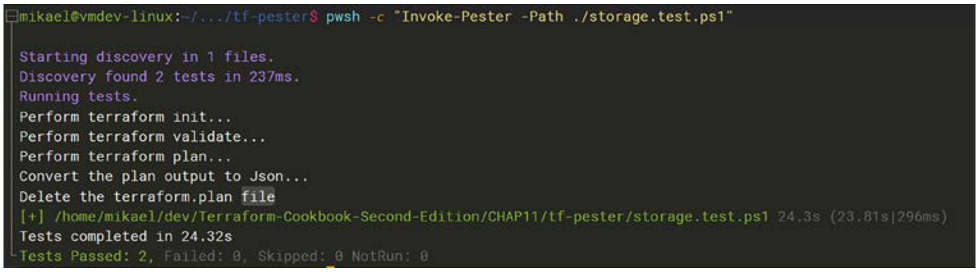
1. The BeforeAll execution
2. The first test that checks the name of the storage
3. The second test that checks whether HTTPS is enabled
4. The AfterAll execution

Here, all tests are failing.

8. So, now, we fix our Terraform configuration by updating the initial configuration with the following configuration:

```
resource "azurerm_storage_account" "storage" {
  name                      = "sademotestpester123"
  resource_group_name       = azurerm_resource_group.rg.name
  location                  = azurerm_resource_group.rg.location
  account_tier              = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = true
}
```

9. And finally, we execute again the Pester command `pwsh -c "Invoke-Pester -Path ./storage.test.ps1"`, and the following image shows the expected output of the test's execution:



```
mikael@vmdev-linux:~/.../tf-pester$ pwsh -c "Invoke-Pester -Path ./storage.test.ps1"

Starting discovery in 1 files.
Discovery found 2 tests in 237ms.
Running tests.
Perform terraform init...
Perform terraform validate...
Perform terraform plan...
Convert the plan output to Json...
Delete the terraform.plan file
[+] /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/tf-pester/storage.test.ps1 24.3s (23.81s|296ms)
Tests completed in 24.32s
Tests Passed: 2, Failed: 0, Skipped: 0 NotRun: 0
```

Figure 11.02: Successful Terraform test with Pester execution

We can see that all tests are running successfully.

How it works...

In *step 1*, we create a new PowerShell file, which will contain our Pester Terraform scripts.

In *step 2*, we start to write this script by adding a `Describe` block, which contains the code for all the tests (called the test suite). For more information about the Pester test file structure, read the documentation here: `https://pester.dev/docs/usage/test-file-structure`.

In *step 3*, we add a `BeforeAll` block, which contains the operations to perform before starting the test suite. Here, in this block, we perform the following operations:

- Run the `terraform init` command.
- Run the `terraform validate` command.
- Run the `terraform plan -out` **terraform.out** command (export the result of `plan` into the `terraform.out` file)
- Run the `terraform show -json terraform.plan` command to show the JSON format of the plan and set the JSON result in the `$plan` PowerShell variable

In *step 4*, we write the first test, which checks the name of the Storage Account that will be provisioned. This script filters the resources `azurerm_storage_account.storage` inside the JSON of the plan, selects the `name` property, and checks whether the name is equal to the expected name.

Refer to this blog post to understand the JSON plan schema: `https://www.scalr.com/blog/opa-series-part-3-how-to-analyze-the-json-plan`.

In *step 5*, we write the second test, which checks the value of the `enable_https_traffic_only` property for the storage account that will be provisioned. This script filters the `azurerm_storage_account.storage` resources inside the JSON of the plan, selects the `enable_https_traffic_only` property, and checks if `enable_https_traffic_only` is equal to `true`.

In *step 6*, we end the `Describe` block by adding the `AfterAll` block, which contains the operations to perform after running the whole test suite. Here, in this block, we delete the `terraform.out` file generated in the `BeforeAll` block (in *step 3*).

Now that we have written the tests, we can run them using Pester.

In *step 7*, we run Pester on the script by executing the `Invoke-Pester -Path ./storage.test.ps1` command.

As the result of this command, if the tests are erroneous, we fix the Terraform configuration according to the test compliance and we run this command above again to check that all tests are running successfully (performed in *steps 8* and *9*).

There's more...

In this sample of Terraform testing with Pester, we stopped the script just after the execution of the tests; we could also have used the `apply` command to add integration tests to this script to check (for example) the connectivity of the storage account after it was created.

For another complete sample with unit and integration tests for the Azure Storage Terraform module, see this GitHub code: `https://github.com/devblackops/presentations/tree/master/PSSummit2021%20-%20Testing%20Terraform%20with%20Pester/demo3/terraform-module-storage`.

Additionally, here is a list of the pros and cons of using the Pester PowerShell Framework:

Pros:

- **Automated testing**: Pester allows you to automate the testing process, so you can easily perform compliance testing on your Terraform code without the need for manual testing.
- **Repeatable and consistent**: With Pester, you can write tests that are repeatable and consistent, which means you can ensure that your code always meets the required compliance standards.
- **Fast feedback loop**: By integrating Pester into your CI/CD pipeline, you can get fast feedback on whether your Terraform code is compliant or not, which can help you identify and fix compliance issues early in the development cycle.

- **Easy to use**: Pester is a PowerShell module, which means you can quickly get started with writing and running tests for your Terraform code.

Cons:

- **Complexity**: Writing tests for Terraform code using Pester can be complex, especially if you need to test complex infrastructure configurations. This can make it difficult to write tests that are both comprehensive and maintainable.
- **Learning curve**: While Pester is relatively easy to use if you're already familiar with PowerShell, there is still a learning curve involved, especially if you are not familiar with PowerShell scripting. This can make it more challenging for teams to adopt Pester for their compliance testing needs.
- **Integration with other tools**: If you are using other tools for your compliance testing, such as specialized compliance frameworks or tools like HashiCorp Sentinel, you may find that integrating Pester into your workflow is more challenging.

See also

- The Pester documentation is available here: `https://pester.dev/`.
- A video that explains how to use Pester to test Terraform configuration: `https://www.youtube.com/watch?v=Sdfxntl6H24`.

Testing the Terraform configuration using Python

In the previous recipe, we learned how to write Terraform tests using a PowerShell script and the Pester framework.

In this recipe, we will learn the same concept for Terraform tests by using the Python language and `pytest` framework.

With `pytest`, the tests are executed just after the `terraform plan` command and before `terraform apply`, so that if the tests are failing, `apply` will not continue.

Let's get started!

Getting ready

To complete this recipe, you'll need to know some basic knowledge of Python.

We need to have Python already installed with pip. The installation documentation for Python is available here: `https://www.python.org/downloads/`. And pip's installation documentation is available here: `https://pip.pypa.io/en/stable/installation/`.

We also install the `pytest` Python test framework by using pip with the `pip install pytest` command.

For more information about `pytest`, read the documentation here: `https://docs.pytest.org/en/7.2.x/contents.html`.

The goal of this recipe is to provide a basic example of how to write and execute tests in `pytest`, which tests to make sure the Azure storage account name is compliant and that the only access to the account is through HTTPS. Exactly as we learned in the previous recipe, the execution of these tests will be performed just after the `terraform plan` command and before `terraform apply`.

In this recipe, we will not detail the steps for writing the Terraform configuration that provisions the Azure Storage Account. The source code for this Terraform configuration is the following:

```
resource "azurerm_storage_account" "storage" {
  name                      = "sademotestpy1"
  resource_group_name       = azurerm_resource_group.rg.name
  location                  = azurerm_resource_group.rg.location
  account_tier              = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = false
}
```

For the demo of the recipe, in this above Terraform configuration, we explicitly configure the storage to be accessible via HTTP.

The complete source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/pytest`.

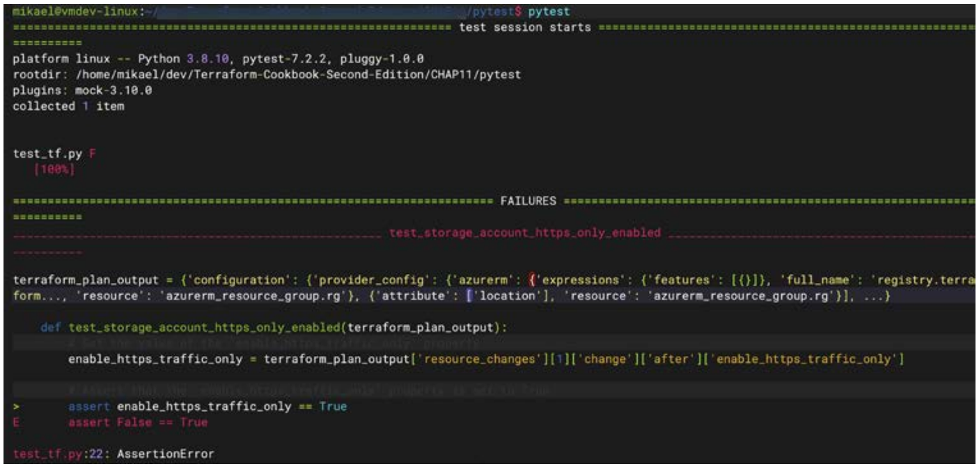
How to do it…

To test the Terraform configuration using `pytest`, perform the following steps:

1. Inside the folder that contains the Terraform configuration, create a new Python file called `test_tf.py` (by default, a `pytest` file must be called `test_<…>.py`).
2. In this Python file, write the following content:

```
import pytest
import subprocess
import json

@pytest.fixture(scope="session")
def terraform_plan_output():
    subprocess.run(["terraform", "init"])
    subprocess.run(["terraform", "plan", "-out", "plan.tfout"])
    show_output = subprocess.check_output(["terraform", "show",
    "-json", "plan.tfout"])
    return json.loads(show_output)

def test_storage_account_https_only_enabled(terraform_plan_output):
    enable_https_traffic_only = terraform_plan_output['resource_
    changes'][1]['change']['after']['enable_https_traffic_only']
    assert enable_https_traffic_only == True
```

3. In the terminal console, execute the `pytest` command.
4. The following image shows the result of this command:



```
mikael@vmdev-linux:~/[redacted]/pytest$ pytest
================================================================ test session starts ==============================================================
==========
platform linux -- Python 3.8.10, pytest-7.2.2, pluggy-1.0.0
rootdir: /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/pytest
plugins: mock-3.10.0
collected 1 item


test_tf.py F
   [100%]

====================================================================== FAILURES ===================================================================
==========
________________________________________________________ test_storage_account_https_only_enabled _________________________________________________
__________

terraform_plan_output = {'configuration': {'provider_config': {'azurerm': {'expressions': {'features': [{}]}, 'full_name': 'registry.terra
form..., 'resource': 'azurerm_resource_group.rg'}, {'attribute': ['location'], 'resource': 'azurerm_resource_group.rg'}], ...}

    def test_storage_account_https_only_enabled(terraform_plan_output):
        [illegible]
        enable_https_traffic_only = terraform_plan_output['resource_changes'][1]['change']['after']['enable_https_traffic_only']

        [illegible]
>       assert enable_https_traffic_only == True
E       assert False == True

test_tf.py:22: AssertionError
```

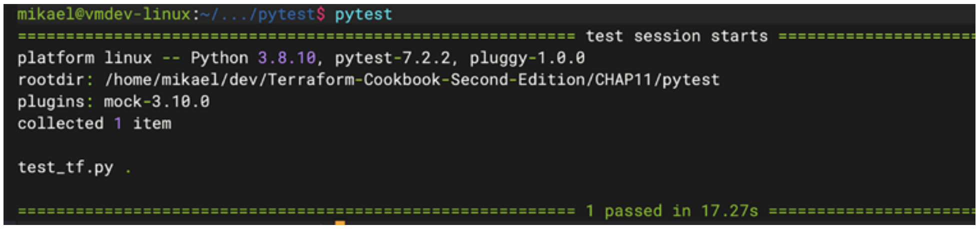
Figure 11.03: pytest execution of the Terraform test with an error

We can see that the assert fails when testing the `enable_https_traffic_only` property.

5. So, now, we fix our Terraform configuration by updating the initial configuration with the following configuration:

```
resource "azurerm_storage_account" "storage" {
  name                     = "sademotestpy123"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = true
}
```

6. And finally, we execute again the Python `pytest` command, and the following image shows the output of the test's execution:



```
mikael@vmdev-linux:~/.../pytest$ pytest
========================================================= test session starts =====================
platform linux -- Python 3.8.10, pytest-7.2.2, pluggy-1.0.0
rootdir: /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/pytest
plugins: mock-3.10.0
collected 1 item

test_tf.py .

========================================================= 1 passed in 17.27s ======================
```

Figure 11.04: Successful Terraform test with pytest execution

All tests pass.

How it works...

In *step 1*, we create a new Python file called `test_tf.py`. This name follows the `pytest` default format, which is `test_<…>.py`.

In *step 2*, we write the `pytest` test code with the following details:

- First, we import a library like `pytest`, the subprocess, and the JSON.
- Then, we write a function `terraform_plan_output`, which runs `terraform init` and `terraform plan` and exports the plan results as JSON in memory.
- Finally, we write a function `test_storage_account_https_only_enabled`, which takes the JSON plan and tests the value of the `enable_https_traffic_only` property with the `assert` `pytest` method.

Refer to this documentation to understand the JSON plan schema: `https://developer.hashicorp.com/terraform/internals/json-format#plan-representation`.

In *step 3*, we run the `pytest` command, which will execute the above `pytest` script.

Looking at the results of this command, if the tests are failing, we fix the Terraform configuration according to the test result and we run this above command again to check that all tests are passing.

There's more…

Here in this recipe, we learned about a basic sample of `pytest` to test Terraform configuration. I encourage you to refer to the `pytest` documentation and also the `tftest` Python library (available here at `https://pypi.org/project/tftest/`) to write more advanced tests.

See also

- The `pytest` documentation is available here: `https://docs.pytest.org/en/7.2.x/contents.html`
- Some posts on Terraform testing with Python: `https://betterprogramming.pub/testing-your-terraform-infrastructure-code-with-python-a3f913b528e3`

  `https://betterprogramming.pub/terraform-resource-testing-101-c9da424faaf3`

  `https://medium.com/saas-infra/terraform-testing-made-easy-with-python-exploring-tftest-925bb207eabd`

Using OPA to check the Terraform configuration

In the previous recipes, we learned how to apply a Terraform compliance check using several tools and languages, such as PowerShell's Pester.

Now, in this recipe, we will study another popular tool, called **Open Policy Agent** (**OPA**), which allows us to perform checks on Terraform configuration.

Before we start, here is a short introduction to OPA.

OPA is an open source, general-purpose policy engine that provides a unified language for managing policies across an organization's software infrastructure. OPA is designed to help organizations define, manage, and enforce policies consistently across different applications, services, and infrastructure.

OPA allows developers and operators to write policies in a declarative language called Rego. Rego is a high-level language that is designed to be easy to read and write, making it accessible to non-experts in policy management. Rego policies can be written to enforce security, compliance, and other operational policies.

OPA is typically integrated into an organization's software infrastructure, where it evaluates policies against data and configuration information. **OPA** can be integrated with various tools and systems, including Kubernetes, Terraform, and Istio, to enforce policies at different stages of the software development and deployment lifecycle.

Some of the key features of **OPA** include:

- **Policy as code**: **OPA** allows policies to be defined and managed using code, making it easier to version-control and track changes.
- **Decoupling policies from code**: **OPA** enables policies to be defined and managed independently of application code, which promotes better separation of concerns and reduces the risk of policy conflicts.
- **Centralized policy management**: **OPA** provides a centralized policy management system, allowing policies to be defined and enforced consistently across different applications and infrastructures.
- **Auditing and traceability**: **OPA** provides an audit trail of policy evaluations, allowing organizations to track and trace policy enforcement over time.

That's all for this short introduction to **OPA**; for more information, read the OPA documentation here: `https://www.openpolicyagent.org/`.

With OPA, tests are executed just after the `terraform plan` command, and before `terraform apply`, so that if the tests are failing, `apply` will not continue.

Let's get started!

Getting ready

To use **OPA**, you need to have the following artifacts:

- **OPA binary**: This is the **OPA** engine that evaluates policies against data and configuration information. To complete this recipe, you'll need to install **OPA** by referring to the installation documentation here: `https://www.openpolicyagent.org/docs/latest/#running-opa`. This is the only requirement for this recipe.

> To install OPA on Windows, go to the open policy agent releases page, available here: https://github.com/open-policy-agent/opa/releases. Once there, click on the desired release (I would recommend the latest). Click on the opa_windows_amd64.exe link on the web page to download the binary executable file.
>
> Rename the downloaded file to opa [in code style] and finally update the PATH [in code style] environment variable to add the downloaded path.

Alternatively, you can use a container image from a registry like Docker Hub.

- **Rego policies**: These are the policies written in the Rego language that OPA evaluates. Rego policies are typically stored in files with a `.rego` extension. You can write Rego policies from scratch or use existing policies from the **OPA** community.

  We will see the Rego policies at the core of this recipe.

- **Data input**: This is the data and configuration information that **OPA** evaluates policies against. Data input can come from a variety of sources, including API requests, configuration files, and data stores.

  In this recipe, we will use JSON as the data input for **OPA**.

- **An OPA query**: This is a request for **OPA** to evaluate a specific policy against given input data. In other words, a query is a question that **OPA** can answer by returning a Boolean value (true or false) based on whether the given input data satisfies the specified policy.

  The core of the recipe is that we will learn about the concept of queries.

In this recipe, we will not detail the steps for writing the Terraform configuration that provisions the Azure Storage Account; the source code for this Terraform configuration is the following:

```
resource "azurerm_storage_account" "storage" {
  name                      = "sademotestopa123"
  resource_group_name       = azurerm_resource_group.rg.name
  location                  = azurerm_resource_group.rg.location
  account_tier              = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = false
}
```

For the demo of the recipe, in this above Terraform configuration, we explicitly configure the storage to be accessible via HTTP and call it `sademotestopa1`.

We will use OPA to check the name of the Azure storage account is compliant and that the security option only allows access to the storage via HTTPS.

The complete source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/opa`.

How to do it...

To use OPA to check the Terraform configuration, perform the following steps:

1. In the same folder that contains the Terraform configuration, create a new file called `sa-policies.rego`, and in this file, start to write the following content:

   ```
   package terraform.policies.storage
   import input as plan
   azurerm_storage[resources] {
         resources := plan.resource_changes[_]
         resources.type == "azurerm_storage_account"
         resources.mode == "managed"
   }
   ```

2. Continue to add the following content:

   ```
   deny[msg] {
     az_storage := azurerm_storage[_]
     r := az_storage.change.after
     not r.enable_https_traffic_only
     msg := sprintf("Storage Account %v must use HTTPS traffic only",
   [az_storage.name])
   }
   ```

3. And add the following content:

   ```
   deny[msg] {
     az_storage := azurerm_storage[_]
     r := az_storage.change.after
     r.name != "sademotestopa123"
   ```

```
    msg := sprintf("Storage Account %v must be named
sademotestopa123", [az_storage.name])
  }
```

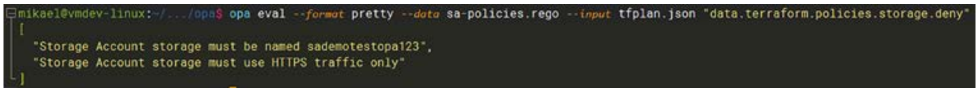
4. Then, inside this folder, run the following Terraform commands:

```
terraform init
terraform plan -out="out.tfplan"
terraform show -json out.tfplan > tfplan.json
```

5. Finally, run the following `opa eval` command:

```
opa eval --format pretty --data sa-policies.rego --input tfplan.json
"data.terraform.policies.storage.deny"
```

The following image shows the output of the result of this command execution:



```
mikael@vmdev-linux:~/.../opa$ opa eval --format pretty --data sa-policies.rego --input tfplan.json "data.terraform.policies.storage.deny"
[
  "Storage Account storage must be named sademotestopa123",
  "Storage Account storage must use HTTPS traffic only"
]
```

Figure 11.05: OPA execution of the Terraform test with an error

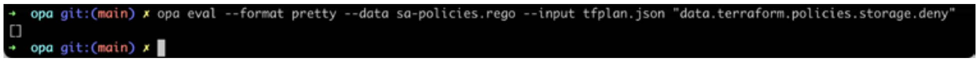
We can see our two policies that are not compliant.

6. Now, we fix the Terraform configuration to be compliant with the policies, with an update via the following configuration:

```
resource "azurerm_storage_account" "storage" {
  name                     = "sademotestopa123"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = true
}
```

7. And we rerun *Step 4* (the Terraform commands) and *step 5* (the `opa eval` command).

The following image shows the output of the `opa eval` command after fixing the configuration:



```
→  opa git:(main) ✗ opa eval --format pretty --data sa-policies.rego --input tfplan.json "data.terraform.policies.storage.deny"
[]
→  opa git:(main) ✗
```

Figure 11.06: Successful OPA execution of the Terraform test

We can see that all policies passed successfully with no errors.

How it works...

In *step 1*, we create a new file, `sa-policies.rego`, which will contain OPA policies that will check the compliance of the Azure storage account that is provisioned.

In this file, we start to define the name of the Rego package name; here, we choose `terraform.policies.storage` as the package name to indicate that it contains Terraform policies on storage.

Then, we add an `input` parameter, which is the result of `terraform plan` in JSON format.

Finally, we declare an `azurerm_storage` function, which filters and gets all `azurerm_storage_account` resources in `terraform plan`'s result.

In *step 2*, we write the first policy called `deny`, which checks that the Azure storage account is configured to be accessible only via HTTPS. The policy code checks the JSON plan content of the `azurerm_storage` object obtained in *step 2* and checks the property `enable_https_traffic_only` in the `after.change` property of the plan. If `enable_https_traffic_only` is `false`, then the policy returns an error message.

In *step 3*, we write the second policy, also called `deny`, which checks that the Azure storage account is named `sademotestopa123`. The policy code checks the JSON plan content of the `azurerm_storage` object obtained in *step 2* and checks the property `name` in the `after.change` property of the plan. If the `name` is not equal to `sademotestopa123`, then the policy fails.

Refer to this blog post to understand the JSON plan schema: `https://www.scalr.com/blog/opa-series-part-3-how-to-analyze-the-json-plan`.

In *Step 4*, we execute the `terraform init` command, then `terraform plan -out="out.tfplan"` to export the plan as a binary file, `out.tfplan`. Finally, we run the `terraform show -json out.tfplan > tfplan.json` command to export the plan in JSON format into the `tfplan.json` file.

In *step 5*, we run the OPA commands to check our Terraform configuration based on OPA's policies. For this, we run the following command:

```
opa eval --format pretty --data sa-policies.rego --input tfplan.json
"data.terraform.policies.storage.deny"
```

With the following arguments:

- The `data` argument is the OPA policy file that we wrote in *steps 1* to *3*
- The `input` argument is the JSON Terraform plan exported in *step 5*
- The last argument is the OPA `query` to check all `deny` policies in the package `terraform.policies.storage`

These 3 arguments represent the OPA artifacts, which are a Rego policy, data input, and an OPA query.

As a result of this command, if the tests are erroneous, we fix the Terraform configuration according to the test output and we run this above command again to check that all tests are passing.

There's more...

In this recipe, we wrote OPA policies, and we ran them directly to check our Terraform configuration.

In real-world scenarios in a company, it is a best practice to test OPA policies before using them.

To test OPA policies, we have 2 possibilities:

- Use the Rego playground, which is a web UI for writing input and testing policies directly in a web browser. The Rego playground is available here: `https://play.openpolicyagent.org/`. This playground can be useful for quickly checking the Rego syntax and execution policies, but it can't be integrated into automatic testing.
- Use an OPA test framework that allows writing Rego tests for Rego policies. The complete documentation is available here: `https://www.openpolicyagent.org/docs/latest/policy-testing/`. The advantages of this solution are that the code for the policy test is stored with the policy, and this policy test can run automatically in CI/CD, for example.

You can also see a test for the Rego policies written in this recipe here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP11/opa/sa-policies_test.rego`.

By using **OPA**, we can run more kinds of tests on Terraform configuration, for example, to do some linting (or static testing) on the Terraform code itself: we can test, for example, that a resource has `lifecycle > ignore_changes` (for some business compliance) and ensure that nobody can delete this `ignore_change` setting.

To do this, we can convert the Terraform configuration into JSON format using the hcl2json tool (available here: `https://www.hcl2json.com/`) and then use the JSON result as input for **OPA**, and write a Rego policy to test the content of the resource code.

The advantage of this method is that we can run this lint test before running `terraform plan` and get linter feedback earlier in the Terraform workflow.

See also

- The **OPA** documentation is available here: `https://www.openpolicyagent.org/`
- The **OPA** for Terraform documentation is available here: `https://www.openpolicyagent.org/docs/latest/terraform/`
- Two videos that explain how to use **OPA** for Terraform, by Ned Bellavance, can be found here:
- Using **OPA** with Terraform - Rego Basics: `https://www.youtube.com/watch?v=DpUDYbFK4IE`
- Open Policy Agent and Terraform - Examining a Terraform Execution Plan with Rego: `https://www.youtube.com/watch?v=YAFICF55aKE`
- The Rego playground: `https://play.openpolicyagent.org/`

Using tfsec to analyze the compliance of Terraform configuration

In the previous recipe, we learned how to use a custom tool to perform an HCL check on the Terraform configuration without running `terraform plan` and exporting the output of the `plan` command.

In this recipe, we will learn how to use the popular tool **tfsec** to analyze the compliance of the Terraform configuration.

`tfsec` (its documentation is available here: `https://aquasecurity.github.io/tfsec/v1.28.1/`) is an open source static analysis tool for Terraform code. It is designed to detect security issues, policy violations, and other potential problems in Terraform code, and provides a set of rules that can be used to scan code for these issues.

`tfsec` works by analyzing the **Abstract Syntax Tree** (**AST**) of Terraform code. This allows it to identify security issues and policy violations based on the structure of the code, without executing the code or connecting to any external services.

Some of the benefits of `tfsec` include:

- **Easy installation**: `tfsec` is easy to install and can be installed using pip or Homebrew, or by downloading a pre-built binary.
- **Customizable rules**: `tfsec` comes with a set of built-in rules, but you can also create your own custom rules using Python.
- **Integration with CI/CD pipelines**: `tfsec` can be easily integrated into CI/CD pipelines using tools like Jenkins, GitLab CI/CD, or CircleCI.
- **Support for multiple Terraform versions**: `tfsec` supports multiple versions of Terraform, including Terraform 0.12, Terraform 0.13, and Terraform 0.14.
- **Detailed output**: `tfsec` provides detailed output, which helps you understand the issues that it has detected in your code. It provides information about the rule that was violated, the location of the violation in the code, and suggestions for how to fix the issue.

`tfsec` is a powerful tool that can help you identify security issues and policy violations in your Terraform code. By using `tfsec` to scan your code, you can ensure that your infrastructure is secure and compliant with best practices and policies.

Let's get started!

Getting ready

To complete this recipe, you will need to install `tfsec` by referring to the installation documentation, which is available here: `https://aquasecurity.github.io/tfsec/v1.28.1/guides/installation/`.

In this recipe, we will not detail the steps for writing the Terraform configuration that provisions the Azure Storage Account; the source code for this Terraform configuration is the following:

```
resource "azurerm_storage_account" "storage" {
  name                     = "sademotesttfsec123"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = false
}
```

For the demo of the recipe, in this above Terraform configuration, we explicitly configure the storage to be accessible via HTTP.

We will use `tfsec` to check the security compliance of this Azure Storage Account.

The complete source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/tfsec`.

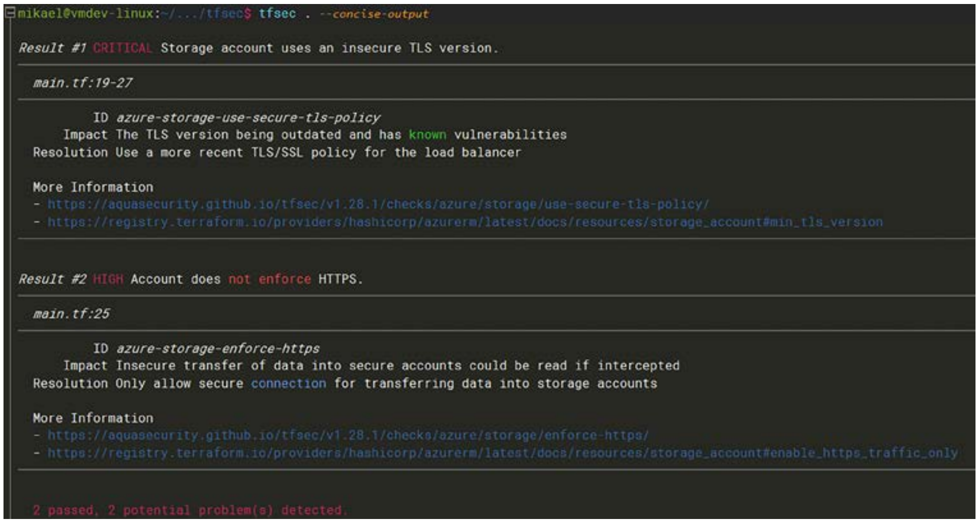
How to do it…

To analyze the Terraform configuration with `tfsec`, perform the following steps:

1. Inside the folder that contains the Terraform configuration to analyze, run the following command:

   ```
   tfsec . --concise-output
   ```

   The following image shows the result of the output of this `tfsec` execution:

   ```
   mikael@vmdev-linux:~/.../tfsec$ tfsec . --concise-output

   Result #1 CRITICAL Storage account uses an insecure TLS version.
   ────────────────────────────────────────────────────────────────
     main.tf:19-27
   ────────────────────────────────────────────────────────────────
             ID azure-storage-use-secure-tls-policy
         Impact The TLS version being outdated and has known vulnerabilities
     Resolution Use a more recent TLS/SSL policy for the load balancer

     More Information
     - https://aquasecurity.github.io/tfsec/v1.28.1/checks/azure/storage/use-secure-tls-policy/
     - https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/storage_account#min_tls_version
   ────────────────────────────────────────────────────────────────


   Result #2 HIGH Account does not enforce HTTPS.
   ────────────────────────────────────────────────────────────────
     main.tf:25
   ────────────────────────────────────────────────────────────────
             ID azure-storage-enforce-https
         Impact Insecure transfer of data into secure accounts could be read if intercepted
     Resolution Only allow secure connection for transferring data into storage accounts

     More Information
     - https://aquasecurity.github.io/tfsec/v1.28.1/checks/azure/storage/enforce-https/
     - https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/storage_account#enable_https_traffic_only
   ────────────────────────────────────────────────────────────────


     2 passed, 2 potential problem(s) detected.
   ```

   Figure 11.07: tfsec execution of the Terraform test with errors

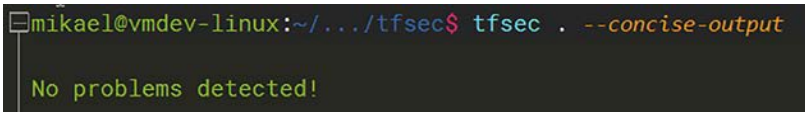
   We can see two failures:

   - Regarding the TLS version
   - Regarding the HTTPS access

2. So, we fix the Terraform configuration with the following code update:

```
resource "azurerm_storage_account" "storage" {
  name                     = "sademotesttfsec123"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = true
  min_tls_version = "TLS1_2"
}
```

3. Then, we rerun the same `tfsec` command that we ran in *step 1*, and we get the following output:



```
mikael@vmdev-linux:~/.../tfsec$ tfsec . --concise-output

No problems detected!
```

Figure 11.08: Successful Terraform test with tfsec execution

We can see that all security problems in the Terraform configuration are fixed.

How it works...

In this recipe, we run the command `tfsec . --concise-output`, where the `.` is the path that contains the Terraform configuration to analyze, and `--concise-output` indicates a concise output.

As a result of this command, if the tests are failing, we fix the Terraform configuration according to the results and we run the above command again to check that all tests pass.

There's more...

In this recipe, we learned about the basic use of `tfsec` via the CLI; the best practice is to integrate this with the CI/CD pipeline, and for more information, we can see an example integration on GitHub Actions here: `https://aquasecurity.github.io/tfsec/v1.28.1/guides/github-actions/github-action/`.

Additionally, in this recipe, we run `tfsec` with built-in rules. We can also write custom rules to provide custom and additional rules. For more information about writing custom rules, refer to the documentation here: `https://aquasecurity.github.io/tfsec/v1.28.1/guides/configuration/custom-checks/`.

Finally, in some cases, we may want to ignore checks on resources to accept the knowledge of the security problems and the risks they pose. Perhaps these rules were too strict for the current context. So, we can add annotations in the Terraform configuration to resources to ignore checks for specific rules. For more information, read the documentation here: `https://aquasecurity.github.io/tfsec/v1.28.1/guides/configuration/ignores/`.

See also

- The documentation for `tfsec` is available here: `https://aquasecurity.github.io/tfsec/v1.28.1/`
- The GitHub repository for `tfsec` is available here: `https://github.com/aquasecurity/tfsec`

Applying Terraform compliance using terraform-compliance

`terraform-compliance` allows you to write tests in a very readable format that follows the idea of **Behavior-Driven Development** (**BDD**).

With `terraform-compliance`, the tests are executed just after the `terraform plan` command, and before `terraform  apply`, so that if the `terraform-compliance` tests are failing, `apply` will continue.

Let's get started!

Getting ready

To complete this recipe, you'll need to install the `terraform-compliance` binary, and for this installation, refer to the installation documentation here: `https://terraform-compliance.com/pages/installation/`.

The requirements for installing `terraform-compliance` are to have already installed `python` and `pip` (see the documentation here: `https://pip.pypa.io/en/stable/installation/`) on your machine.

The goal of this recipe is to write and execute compliance tests for the Terraform configuration that provisions an Azure Storage Account.

The compliance rules check that the storage is only accessible in `HTTPS` mode and has the `DEMO = book` tag.

The Terraform configuration that provisions the Azure storage account is already complete as follows:

```
resource "azurerm_storage_account" "storage" {
  name                     = "sademotestcomp123"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = false
}
```

For the demo of the recipe, in this above Terraform configuration, we explicitly configure the storage to be accessible via HTTP and have no tags.

Now in the recipe, we will write and execute `terraform-compliance` to check the configuration, and we will fix the configuration to conform with the compliance rules.

The complete source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/tf-compliance`.

How to do it...

Perform the following steps to execute `terraform-compliance` on the Terraform configuration:

1. Inside the folder that contains the Terraform configuration, to provision the Azure Storage account, create a new folder called `acceptance`.
2. Inside this `acceptance` folder, create a new file called `storage.feature`.
3. At the top of this new file, add the following content:

    ```
    Feature: Test compliance of Azure Storage Account
    ```

4. Add the first test by adding the following content to this file:

    ```
      Scenario: Ensure our Azure Storage have Tag DEMO with value demo
        Given I have azurerm_storage_account defined
        Then it must contain tags
        Then it must contain DEMO
        And its value must be "book"
    ```

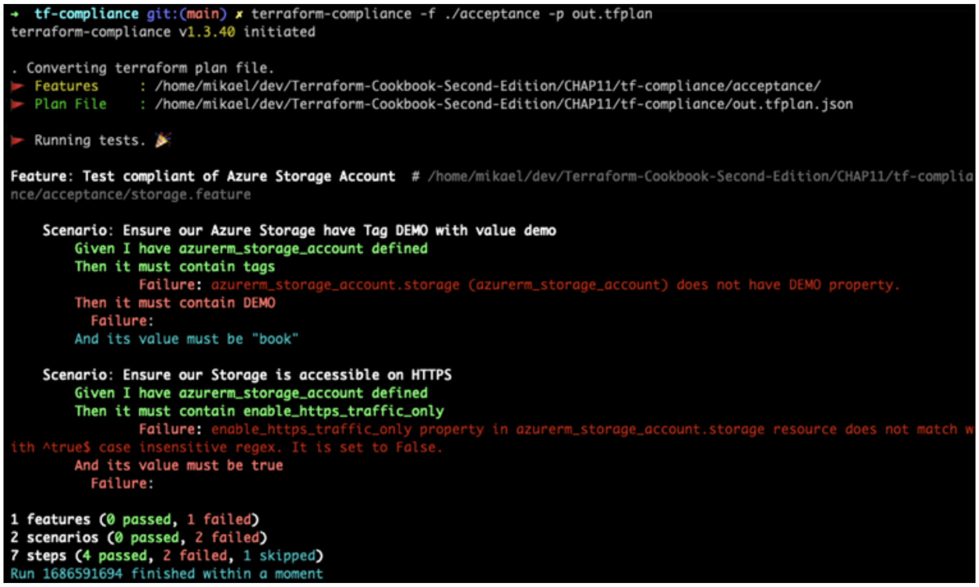
5. In this file, add the second test by adding the following content:

```
    Scenario: Ensure our Storage is accessible via HTTPS only
      Given I have azurerm_storage_account defined
      Then it must contain enable_https_traffic_only
      And its value must be true
```

6. Inside the Terraform configuration folder, run the command `terraform init`.
7. Then, run the Terraform command `terraform plan -out="out.tfplan"`.
8. Run the following `terraform-compliance` command:

```
terraform-compliance -f ./acceptance -p out.tfplan
```

The following image shows the output of this command:



```
➜  tf-compliance git:(main) ✗ terraform-compliance -f ./acceptance -p out.tfplan
terraform-compliance v1.3.40 initiated

. Converting terraform plan file.
▶ Features     : /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/tf-compliance/acceptance/
▶ Plan File    : /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/tf-compliance/out.tfplan.json

▶ Running tests. 🎉

Feature: Test compliant of Azure Storage Account  # /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/tf-complia
nce/acceptance/storage.feature

    Scenario: Ensure our Azure Storage have Tag DEMO with value demo
        Given I have azurerm_storage_account defined
        Then it must contain tags
                Failure: azurerm_storage_account.storage (azurerm_storage_account) does not have DEMO property.
        Then it must contain DEMO
          Failure:
        And its value must be "book"

    Scenario: Ensure our Storage is accessible on HTTPS
        Given I have azurerm_storage_account defined
        Then it must contain enable_https_traffic_only
                Failure: enable_https_traffic_only property in azurerm_storage_account.storage resource does not match w
ith ^true$ case insensitive regex. It is set to False.
        And its value must be true
          Failure:

1 features (0 passed, 1 failed)
2 scenarios (0 passed, 2 failed)
7 steps (4 passed, 2 failed, 1 skipped)
Run 1686591694 finished within a moment
```

Figure 11.09: terraform-compliance execution of the Terraform test with errors

The tests will fail.

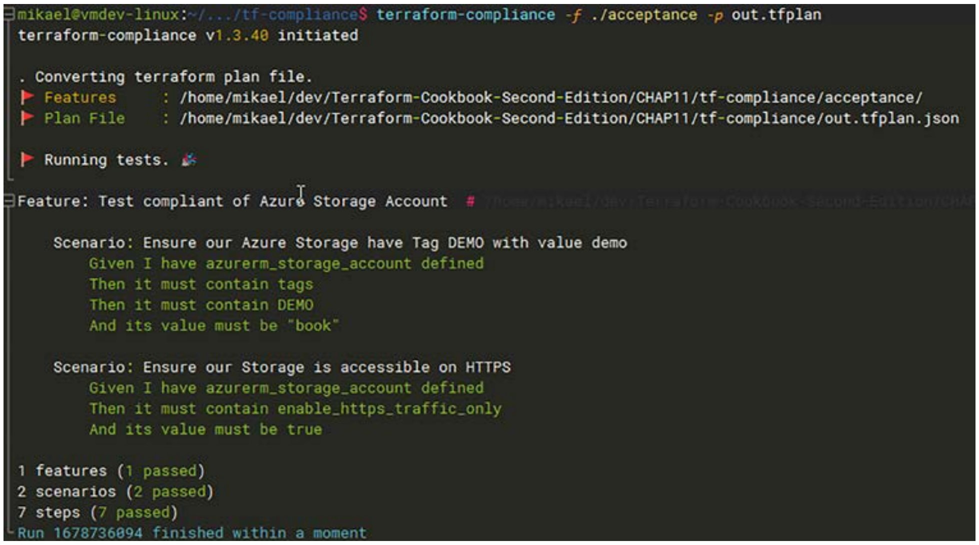
9. Fix the Terraform configuration by updating the Azure Storage configuration as follows:

```
resource "azurerm_storage_account" "storage" {
  name                         = "sademotestcomp123"
  resource_group_name          = azurerm_resource_group.rg.name
  location                     = azurerm_resource_group.rg.location
```

```
    account_tier             = "Standard"
    account_replication_type = "GRS"
    enable_https_traffic_only = true
    tags = {
      DEMO = "book"
    }
  }
```

10. Rerun the Terraform command `terraform plan -out="out.tfplan"`.
11. Finally, rerun the `terraform-compliance` command `terraform-compliance -f ./acceptance -p out.tfplan`.

    The following image shows the output of the `terraform-compliance` command execution when all tests are fixed.



```
mikael@vmdev-linux:~/.../tf-compliance$ terraform-compliance -f ./acceptance -p out.tfplan
terraform-compliance v1.3.40 initiated

. Converting terraform plan file.
▶ Features     : /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/tf-compliance/acceptance/
▶ Plan File    : /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHAP11/tf-compliance/out.tfplan.json

▶ Running tests.

Feature: Test compliant of Azure Storage Account  # /home/mikael/dev/Terraform-Cookbook-Second-Edition/CHA

    Scenario: Ensure our Azure Storage have Tag DEMO with value demo
        Given I have azurerm_storage_account defined
        Then it must contain tags
        Then it must contain DEMO
        And its value must be "book"

    Scenario: Ensure our Storage is accessible on HTTPS
        Given I have azurerm_storage_account defined
        Then it must contain enable_https_traffic_only
        And its value must be true

1 features (1 passed)
2 scenarios (2 passed)
7 steps (7 passed)
Run 1678736094 finished within a moment
```

Figure 11.10: Successful execution of terraform-compliance for the Terraform test

    All tests are executed successfully.

12. Run the `terraform apply` command to apply changes.

How it works…

In *step 1*, we create a new folder called `acceptance` inside the folder that contains our Terraform configuration. Inside this folder, we create a new file called `storage.feature`. This file will contain all the test specifications that will be written in this recipe.

In *step 2*, we initialize this file by adding the `Feature` title `Test compliance of Azure Storage Account`. In `terraform-compliance`, a `Feature` is a logical group of tests, also called a test suite.

In *step 3*, we add in this file the code for the first scenario test, which checks that the Azure storage account has the tag `DEMO = book`.

The language of the scenario is based on BDD with `GIVEN`, (optional `WHEN`), and `THEN` instructions.

In `GIVEN`, we specify the initial condition; here, the condition is that the Terraform configuration contains the `azurerm_storage_account` resource. In the `THEN` instruction, we specify what is expected; here we expect that storage contains a `Tag` property with `DEMO=book` as the key value.

In *Step 4*, we add the second scenario test, which checks that the Azure storage account allows only HTTPS access.

In `GIVEN`, we specify the initial condition; here, the condition is that the Terraform configuration contains the `azurerm_storage_account` resource. In `THEN`, we specify what is expected; here we expect that the storage contains an `enable_https_traffic_only` property with a value of `true`.

For more information about the BDD `terraform-compliance` reference, read the documentation here: `https://terraform-compliance.com/pages/bdd-references/`

Now that we have written the test code, we will run `terraform-compliance` during the Terraform workflow.

In *Step 5*, we start the Terraform workflow by running the `terraform init` command.

In *Step 6*, we execute the command `terraform plan -out=out.tfplan` to export the plan result into the file `out.tfplan`.

In *step 7*, we run the `terraform-compliance` command by specifying the following arguments:

- `-f` : The folder path where we wrote the acceptance tests in *steps 2* to *4*
- `-p`: The `terraform plan` result exported in *step 6*

We get two test failures: first due to the absence of the tag and the second due to the `enable_https_only` property set to `false`.

Then, in *step 8*, we fix the Terraform configuration, and in *step 9*, we run the above workflow again with the `terraform init`, the `terraform plan`, and the `terraform-compliance` execution.

By the end, in *step 11*, we can apply changes by running `terraform apply`.

There's more...

For more information about how to test compliance with `terraform-compliance` on Azure Terraform configurations, read the documentation here: `https://learn.microsoft.com/en-us/azure/developer/terraform/best-practices-compliance-testing`.

Additionally, in this recipe, we learned how to execute the CLI of `terraform-compliance` manually. In a business scenario, the next goal is to integrate it with a CI/CD pipeline like Jenkins, Azure DevOps, and so on.

See also

- The documentation for `terraform-compliance` is available here: `https://terraform-compliance.com/`
- A post about `terraform-compliance` is available here: `https://dev.to/aws-builders/trusting-in-your-iac-terraform-compliance-4cch`
- The Python package source for `terraform-compliance` is available here: `https://pypi.org/project/terraform-compliance/`
- Azure documentation on integration tests using `terraform-compliance`: `https://learn.microsoft.com/en-us/azure/developer/terraform/best-practices-compliance-testing`

Testing Terraform module code with Terratest

When developing a Terraform module that will be used in multiple Terraform configurations and shared with other teams, there is one step that is often neglected and that is the testing of the module.

Among the Terraform framework and testing tools is the **Terratest** framework, and the documentation is available here – `https://gruntwork.io/` – which is popular and allows you to write tests in the Go language.

In this recipe, we will study how to use Terratest to write and run integration tests against Terraform configuration and modules.

Let's get started!

Getting ready

The **Terratest** test framework is written in Go. That's why, as a prerequisite, we need to install Go by going to `https://golang.org/`.

The minimum Go version required for Terratest is specified here: `https://terratest.gruntwork.io/docs/getting-started/quick-start/#requirements`.

The Go installation depends on your operating system, and the installation documentation is available here: `https://go.dev/doc/install`.

The goal of this recipe is to write the integration tests for a very simple module, which we will also write in this recipe to serve as a demonstration.

The source code for this chapter with the module and its test is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/testing-terratest`.

How to do it...

This recipe is in two parts: the first part concerns the writing of the module and its tests, and the second part concerns the execution of the tests.

To write the module and its tests, we perform the following steps:

1. We create a new `module` folder that will contain the Terraform configuration for the module. In this `module` folder, we create a `main.tf` file, which contains the following code:

```
variable "string1" {}

variable "string2" {}

## PUT YOUR MODULE CODE
##_____________________
output "stringfct" {
  value = format("This is test of %s with %s", var.string1,
upper(var.string2))
}
```

2. In this `module` folder, we create the `fixture` folder inside a `tests` folder.
3. Then, in this `fixture` folder, we create a `main.tf` file, which contains the following Terraform configuration:

```
module "demo" {
  source  = "../../"
  string1 = "module"
  string2 = "terratest"
}

output "outmodule" {
  value = module.demo.stringfct
}
```

4. In the `test` folder, we create a module _`test.go` file, which contains the following code:

```
package test

import (
   "testing"
   "github.com/gruntwork-io/terratest/modules/terraform"
   "github.com/stretchr/testify/assert"
)

func TestTerraformModule(t *testing.T) {
   terraformOptions := &terraform.Options{
      // path to the terraform configuration
      TerraformDir: "./fixture",
   }

   // lean up resources with "terraform destroy" at the end of the
test.
   defer terraform.Destroy(t, terraformOptions)

   // Run "terraform init" and "terraform apply". Fail the test if
there are any errors.
   terraform.InitAndApply(t, terraformOptions)

   // Run 'terraform output' to get the values of output variables
and check they have the expected values.
   output := terraform.Output(t, terraformOptions, "outmodule")
   assert.Equal(t, "This is test of module with TERRATEST", output)
}
```

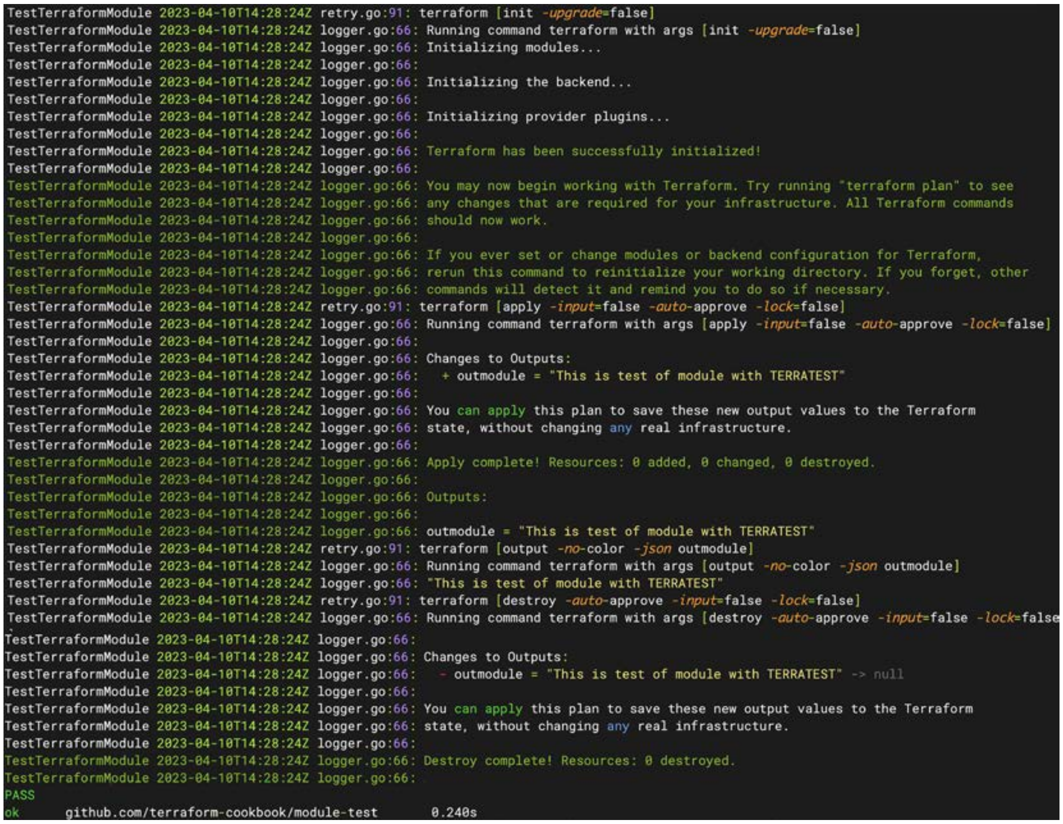
5. To execute the tests, we execute the `go test` command as follows:

```
go test -v
```

During this execution, Terratest will carry out the following actions in order:

- Execute the `terraform init` and `terraform apply` commands on the Terraform test code located in the `fixture` folder.
- Get the value of the `outmodule` output.
- Compare this value with the expected value.
- Execute the `terraform destroy` command.
- Display the test results.

The next screenshot shows the execution of the tests on our module:



```
TestTerraformModule 2023-04-10T14:28:24Z retry.go:91: terraform [init -upgrade=false]
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Running command terraform with args [init -upgrade=false]
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Initializing modules...
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Initializing the backend...
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Initializing provider plugins...
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Terraform has been successfully initialized!
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: You may now begin working with Terraform. Try running "terraform plan" to see
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: any changes that are required for your infrastructure. All Terraform commands
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: should now work.
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: If you ever set or change modules or backend configuration for Terraform,
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: rerun this command to reinitialize your working directory. If you forget, other
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: commands will detect it and remind you to do so if necessary.
TestTerraformModule 2023-04-10T14:28:24Z retry.go:91: terraform [apply -input=false -auto-approve -lock=false]
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Running command terraform with args [apply -input=false -auto-approve -lock=false]
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Changes to Outputs:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:   + outmodule = "This is test of module with TERRATEST"
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: You can apply this plan to save these new output values to the Terraform
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: state, without changing any real infrastructure.
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Outputs:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: outmodule = "This is test of module with TERRATEST"
TestTerraformModule 2023-04-10T14:28:24Z retry.go:91: terraform [output -no-color -json outmodule]
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Running command terraform with args [output -no-color -json outmodule]
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: "This is test of module with TERRATEST"
TestTerraformModule 2023-04-10T14:28:24Z retry.go:91: terraform [destroy -auto-approve -input=false -lock=false]
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Running command terraform with args [destroy -auto-approve -input=false -lock=false
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Changes to Outputs:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:   - outmodule = "This is test of module with TERRATEST" -> null
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: You can apply this plan to save these new output values to the Terraform
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: state, without changing any real infrastructure.
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66: Destroy complete! Resources: 0 destroyed.
TestTerraformModule 2023-04-10T14:28:24Z logger.go:66:
PASS
ok      github.com/terraform-cookbook/module-test       0.240s
```

Figure 11.11: Execution of Terraform test with Terratest

You can see, in this screenshot, the different operations that have been executed by the `go test -v` command, as well as the result of the tests.

How it works...

In the first part of this recipe, we worked on the development of the module and its tests with the Terratest framework. We wrote the module's code, which focuses on the module's output. In *step 3*, in the `fixture` folder, we wrote Terraform configuration that uses the module locally and that we will use to test it.

What is important in this configuration is to have output in the module. In Terratest, we will use the output to test that the module returns the expected value.

In *step 4*, we write the module tests in Go. The code is composed as follows: in the first lines of this code, we import the libraries needed to run the tests, including the `terratest` and `assert` libraries. Then, we create a `TestTerraformModule` function, which takes `*testing.T` as a parameter, which indicates that it is a test function.

The following are the details of the code of this function, which is composed of five lines of code.

In the first line, we define the test options with the folder containing the Terraform configuration that will be executed during the tests:

```
terraformOptions := &terraform.Options{
   // path to the terraform configuration
   TerraformDir: "./fixture",
}
```

Then, we define the `terraform.Destroy` function, which allows us to execute the `terraform destroy` command at the end of the tests, as described in the following code:

```
defer terraform.Destroy(t, terraformOptions)
```

Then, we call the `terraform.InitAndApply` function, which allows us to execute the `terraform init` and `apply` commands, as described in the following code:

```
terraform.InitAndApply(t, terraformOptions)
```

After executing the `InitAndApply` command, we will retrieve the value of the output, which is called `outmodule`:

```
output := terraform.Output(t, terraformOptions, "outmodule")
```

Finally, we use `assert` to test the previously recovered value of the output with the value we expect:

```
assert.Equal(t, "This is test of module with TERRATEST", output)
```

Then, in the second part of this recipe, we work on the execution of the tests. The first step is to initialize the Go package by navigating to the `test` folder.

Then, running the `go mod init <package name>` command, here, in this sample, we choose the package name `github.com/terraform-cookbook/module-test`, and run the `go mod tidy` command.

These two above commands generate `go.mod` and `go.sub` files, which contain a list of all the Go package dependencies.

Then, inside the `test` folder, we run the tests by executing this command:

```
go test -v
```

There's more...

Terratest allows you to execute integration tests on Terraform configuration that allow you to provision resources, execute the tests, and finally destroy the resources.

We have seen in the prerequisites for this recipe that the setup of the Golang development environment requires actions that vary from one operating system to another. To facilitate this task, you can execute your **Terratest** tests in a Docker container that already has an environment configured. The `Dockerfile` corresponding to this container is available here: `https://austincloud.guru/2021/06/24/running-terratest-in-a-docker-container/`.

> If Terraform modules provide resources via cloud providers, the authentication parameters must be set before running tests.

Finally, as said in the introduction to this recipe, **Terratest** is not limited to Terraform—it also allows testing on Packer, Docker, and Kubernetes code. But it goes further by also running tests against cloud providers such as AWS, Azure, and GCP.

The following code snippet shows how to test the size of the VM in Azure based on Terraform's output:

```
azureVmName := terraform.Output(t, terraformOptions, "vm_name")
resourceGroupName := terraform.Output(t, terraformOptions, "rg_name")
```

```
actualVMSize := azure.GetSizeOfVirtualMachine(t, vmName,
resourceGroupName, "")
expectedVMSize := compute.VirtualMachineSizeTypes("Standard_DS2_v2")
```

See also

- Terratest's official website is available here: `https://terratest.gruntwork.io/`
- Terratest's documentation is available here: `https://terratest.gruntwork.io/docs/`
- Example Terratest code is available here: `https://github.com/gruntwork-io/terratest/tree/master/examples`
- Read this great article about Terratest: `https://blog.octo.com/en/test-your-infrastructure-code-with-terratest/`

Testing the Terraform configuration using Kitchen-Terraform

We have already studied, in the *Testing Terraform module code with Terratest* recipe of this chapter, how to test Terraform modules using the **Terratest** framework.

In this recipe, we will test Terraform configuration using another tool: `KitchenCI` and its **Kitchen-Terraform** plugin.

Getting ready

`Kitchen-Terraform` is written in Ruby and is a plugin for `KitchenCI` (more simply called Kitchen), which is an IaC testing tool. To apply this recipe properly, you must first understand the principles and workflow of Kitchen, documented at `https://kitchen.ci/index.html`.

As Kitchen is written in Ruby, you will need to install Ruby (available at `https://www.ruby-lang.org/en/`) on your computer by following the installation documentation available at `https://www.ruby-lang.org/en/documentation/installation/`.

In addition to Ruby, we need to install Bundler, available from `https://bundler.io/`. This is the package manager for Ruby packages by running the commannd `gem install bundler`.

We can install `kitchen-terraform` by using the method recommended by Kitchen using gems and bundles by following this procedure:

1. In the folder that contains the Terraform configuration to be tested, we create a `Gemfile` that contains the list of packages (here, we specify the `kitchen-terraform` package) to install, containing the following:

   ```
   source "https://rubygems.org/" do
     gem "kitchen-terraform"
   end
   ```

2. In a terminal, execute the following command to install the packages referenced in the Gemfile (in Linux, run on `sudo` mode to have all the necessary permissions to install packages):

   ```
   bundle install
   ```

The execution of the preceding command installs all the packages necessary to run `Kitchen-Terraform`.

If you have any issues with the kitchen-terraform installation, read the documentation here: https://github.com/newcontext-oss/kitchen-terraform#kitchen-terraform-ruby-gem

Finally, concerning the writing of the tests, we will use **Inspec**, which is a test framework based on **Rspec**. **Inspec** allows you to test local systems or even infrastructure in the cloud. For more information about **Inspec**, I suggest you read its documentation at `https://www.inspec.io/`.

In this recipe, the goal is not to test the creation of the network and the VMs, only the inventory file.

Finally, as with all integration testing, it is preferable to have an isolated system or environment to run the tests.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/kitchen`.

How to do it...

To test the Terraform configuration execution with `Kitchen-Terraform`, perform the following steps:

1. Inside the folder containing the Terraform configuration, create the **Inspec** `test` folder with the following tree:

   ```
   test > integration > kt_suite
   ```

2. In this `kt_suite` folder, add the **Inspec** profile file named `inspec.yml` with the following content:

```
---
name: default
```

3. In the `kt_suite` folder, create a new folder called `controls`, which will contain the **Inspec** tests. Then, inside the `controls` folder, add a new `inventory.rb` file with the following content:

```
control "check_inventory_file" do
  describe file('./inventory') do
    it { should exist }
    its('size') { should be > 0 }
  end
end
```

4. At the root of the Terraform configuration folder, we create a Kitchen configuration file called `kitchen.yml` with the following content:

```
---
driver:
  name: terraform

provisioner:
  name: terraform

verifier:
  name: terraform
  systems:
    - name: basic
      backend: local
      controls:
        - check_inventory_file
platforms:
  - name: terraform
suites:
  - name: kt_suite
```

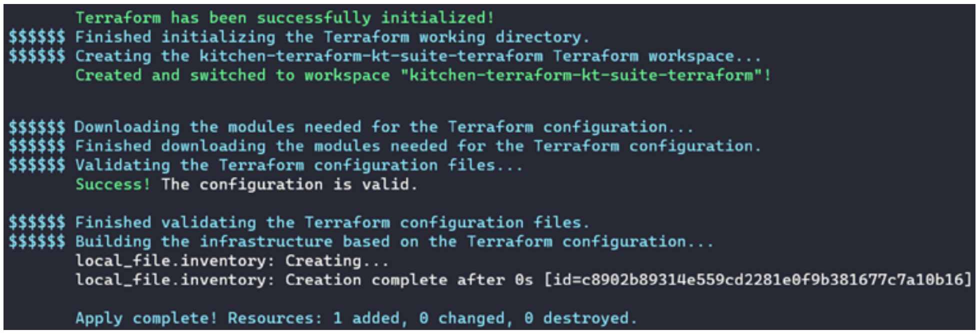
5. In a terminal (running in the root of the Terraform configuration folder), run the following `kitchen` command:

```
kitchen test
```

The result of this execution is shown in the following three screenshots.

This execution takes place in the same console and the same workflow. I've split this into three screenshots for better visibility because you can't see everything on just one screen.

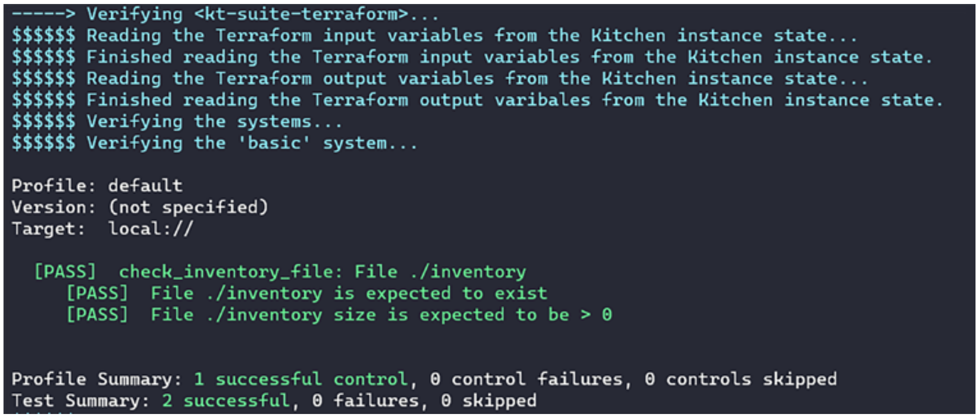
The following screenshot shows the execution of the `init` and `apply` commands:



```
       Terraform has been successfully initialized!
$$$$$$ Finished initializing the Terraform working directory.
$$$$$$ Creating the kitchen-terraform-kt-suite-terraform Terraform workspace...
       Created and switched to workspace "kitchen-terraform-kt-suite-terraform"!


$$$$$$ Downloading the modules needed for the Terraform configuration...
$$$$$$ Finished downloading the modules needed for the Terraform configuration.
$$$$$$ Validating the Terraform configuration files...
       Success! The configuration is valid.

$$$$$$ Finished validating the Terraform configuration files.
$$$$$$ Building the infrastructure based on the Terraform configuration...
       local_file.inventory: Creating...
       local_file.inventory: Creation complete after 0s [id=c8902b89314e559cd2281e0f9b381677c7a10b16]

       Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Figure 11.12: Execution of init and apply for a Terraform test with Kitchen-Terraform

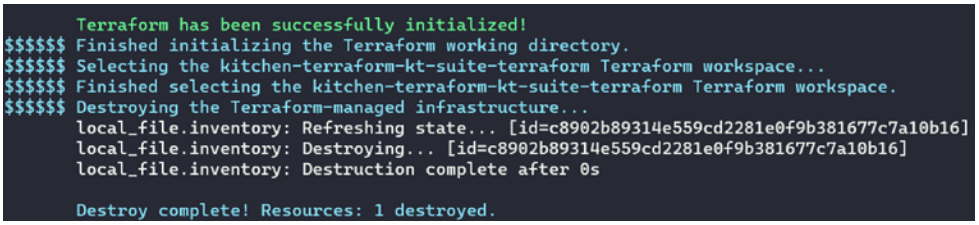
The following screenshot shows the execution of **Inspec**:



```
-----> Verifying <kt-suite-terraform>...
$$$$$$ Reading the Terraform input variables from the Kitchen instance state...
$$$$$$ Finished reading the Terraform input variables from the Kitchen instance state.
$$$$$$ Reading the Terraform output variables from the Kitchen instance state...
$$$$$$ Finished reading the Terraform output varibales from the Kitchen instance state.
$$$$$$ Verifying the systems...
$$$$$$ Verifying the 'basic' system...

Profile: default
Version: (not specified)
Target:  local://

  [PASS]  check_inventory_file: File ./inventory
     [PASS]  File ./inventory is expected to exist
     [PASS]  File ./inventory size is expected to be > 0


Profile Summary: 1 successful control, 0 control failures, 0 controls skipped
Test Summary: 2 successful, 0 failures, 0 skipped
```

Figure 11.13: Execution of Inspec for a Terraform test

This last screenshot shows the `destroy` command:



```
       Terraform has been successfully initialized!
$$$$$$ Finished initializing the Terraform working directory.
$$$$$$ Selecting the kitchen-terraform-kt-suite-terraform Terraform workspace...
$$$$$$ Finished selecting the kitchen-terraform-kt-suite-terraform Terraform workspace.
$$$$$$ Destroying the Terraform-managed infrastructure...
       local_file.inventory: Refreshing state... [id=c8902b89314e559cd2281e0f9b381677c7a10b16]
       local_file.inventory: Destroying... [id=c8902b89314e559cd2281e0f9b381677c7a10b16]
       local_file.inventory: Destruction complete after 0s

       Destroy complete! Resources: 1 destroyed.
```

Figure 11.14: Execution of destroy for a Terraform test with Inspec

These three screenshots show the execution of Terraform; the successful execution of the **Inspec** tests, which indicates that my inventory file was indeed generated by Terraform; and finally the destruction of the resources that had been allocated for the tests.

How it works...

In *steps 1* to *3*, we write the inspection tests with the following steps:

1. First, we create the folder tree that will contain the profile and the **Inspec** tests. In the `kt_suite` folder, we create the `inspec.yml` file, which is the **Inspec** profile. In our case, this just contains the `name` property with the `default` value.

   To learn more about Inspec profiles, refer to the documentation at `https://www.inspec.io/docs/reference/profiles/`.

2. Then, in the `controls > inventory.rb` file, we write the Inspec tests (in Rspec format) by creating a control "`check_inventory_file`" that will contain the tests. In these tests, we use the resource file Inspec (see the documentation at `https://www.inspec.io/docs/reference/resources/file/`), which allows us to run tests on files. Here, the property of this resource is `inventory`, which is the name of the inventory file generated by Terraform. In this control, we have written two tests:

   - `it { should exist }`: This inventory file must exist on disk.
   - `its('size') { should be > 0 }`: The size of this file must be `> 0`, so it must contain some content.

Once we have written the tests, in *step 4*, we create the `kitchen.yml` file, which contains the Kitchen configuration, consisting of three parts, the first one being the driver:

```
driver:
  name: terraform
```

The driver is the platform that is used for testing. Kitchen supports a multitude of virtual and cloud platforms. In our case, we use the `terraform` driver provided by the `Kitchen-Terraform` plugin.

The documentation on the drivers supported by Kitchen is available at `https://kitchen.ci/docs/drivers/`.

The second part of the `kitchen.yml` file is the provisioner:

```
provisioner:
  name: terraform
```

The provisioner is the tool that will configure the VMs. It can use scripts, Chef, Ansible, or **Desired State Configuration** (**DSC**). In our case, since in our test, we don't provision VMs, we use the `terraform` provisioner provided by `Kitchen-Terraform`.

The documentation on Kitchen-supported provisioners is available at `https://kitchen.ci/docs/provisioners/`.

The third part is the verifier:

```
verifier:
  name: terraform
  systems:
    - name: basic
      backend: local
      controls:
        - check_inventory_file
platforms:
  - name: terraform
suites:
  - name: kt_suite
```

The verifier is the system that will test the components applied by the provisioner. We can use **Inspec**, Chef, Shell, or Pester as our testing framework. In our case, we configure the verifier on the control and the **Inspec** test suite we wrote in *step 2*. In addition, the control property is optional – it allows us to filter the **Inspec** controls to be executed during the tests.

Documentation on Kitchen-supported `verifiers` is available at `https://kitchen.ci/docs/verifiers/`.

3. Finally, in the last step, we perform the tests by executing the `kitchen test` command, which, based on the YAML kitchen configuration, will perform the following actions:
4. Execute the `init` and `apply` commands of the Terraform workflow.
5. Run the Inspec tests.
6. Execute the `destroy` Terraform command to delete all resources provisioned for the test.

There's more…

To go deeper into the writing of the tests, we could have added the `Inspec its('content')` expression, which allows us to test the content of the file, as explained in the Inspec documentation at `https://www.inspec.io/docs/reference/resources/file/`.

Concerning the execution of the tests in this recipe, we must execute the `kitchen test` command. In the case of integration tests in which, after executing the tests, we don't want to destroy the resources that have been built with Terraform, we can execute the `kitchen verify` command.

Finally, as mentioned in the introduction, in this recipe we used `Kitchen-Terraform` to test a Terraform configuration, but we can also use it to test Terraform modules.

See also

- KitchenCI's documentation is available at `https://kitchen.ci/`
- The source code for the `Kitchen-Terraform` plugin is available on GitHub at `https://github.com/newcontext-oss/kitchen-terraform`
- You can find tutorials on `Kitchen-Terraform` at `https://newcontext-oss.github.io/kitchen-terraform/tutorials/`
- For more information about the `kitchen test` command, see the documentation at `https://kitchen.ci/docs/getting-started/running-test/`

Using the new integrated Terraform module integration test

In the previous recipe, we learned how to perform unit tests on a Terraform module using the Terratest framework written in Go.

Since Terraform v1.0, HashiCorp has introduced a test integration feature for modules.

At the time of writing, the new test feature has been changed in the alpha version of Terraform 1.6.0. See here for more information: https://github.com/hashicorp/terraform/releases/tag/v1.6.0-alpha20230816

In this recipe, we will learn how to use the integrated Terraform module integration test.

Let's get started!

Getting ready

For this recipe, no specific software requirements are required; we will use only Terraform configuration and the Terraform CLI.

To complete this recipe, we have already written one Terraform module, which provisions an Azure resource group and one Azure storage account.

The following source code is the main code for this Terraform module:

```
resource "azurerm_storage_account" "storage" {
  name                     = "sademotest1"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = false
}
output "https_enabled" {
  value = azurerm_storage_account.storage.enable_https_traffic_only
}

output "storage_name" {
  value = azurerm_storage_account.storage.name
}
```

The complete source code for the module is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/moduletest`.

The goal of this recipe is to apply security and naming compliance to ensure that our Azure storage account has the following requirements:

- The storage account is configured to allow only HTTPS.
- The name of the Azure account must end with *123*.

For the execution of this recipe, we explicitly write code for the module that does not respect the above requirements.

Now, in this recipe, we will write tests to check the module requirements, the tests result will fail (in red), then we will fix the module code to respect compliance, and finally, we will re-run those tests and check that all are passing (green).

This test execution method is called **Test-Driven Design** (**TDD**); for more information, read this article from Packt here: `https://subscription.packtpub.com/book/web-development/9781782174929/1/ch01lvl1sec09/understanding-test-driven-development`.

How to do it...

To operate integrated tests on this Terraform module, perform the following steps:

1. Inside the `moduletest` folder, create a `tests` folder, and inside this folder, create a `defaults` folder. Inside the `defaults` folder, create a new file, which we call `test_defauts.tf`. In this file, start by writing the following Terraform configuration:

```
terraform {
  required_providers {
    test = {
      source = "terraform.io/builtin/test"
    }
  }
}
module "storage" {
  source = "../.."
}
```

2. In this file, add the following Terraform configuration to test HTTPS access:

```
resource "test_assertions" "https" {
  component = "https"
  equal "scheme" {
    description = "https must be enabled"
    got         = module.storage.https_enabled
    want        = true
  }
}
```

3. Continue by adding this Terraform configuration in this file to test the Azure storage account name:

```
resource "test_assertions" "storageName" {
  component = "name"
  check "storage_name" {
    description = "storage name must end with 123"
    condition   = can(regex("^123", module.storage.storage_name))
  }
}
```

4. Now, inside the `moduletest` folder, run the following Terraform command:

```
terraform test
```

5. Then, we will fix the Terraform configuration of the module with the following code (see the highlighted, updated code):

```
resource "azurerm_storage_account" "storage" {
  name                     = "sademotest123"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = azurerm_resource_group.rg.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
  enable_https_traffic_only = true
}
```

6. Finally, we rerun the `terraform test` command.

How it works...

In *step 1*, we create the Terraform test folder structure with the following hierarchy `moduletest > tests > defaults`. In this `defaults` folder, we will create a new Terraform file `test_default.tf`.

In this file, we add code for the following actions:

- Use the `terraform.io/builtin/test` Terraform provider
- Reference our module using the `module` block and internal `source`

In *step 2*, in this file, we add a Terraform resource `test_assertions` that tests the HTTPS requirement by checking that the `https_enabled` module output is equal to `true`.

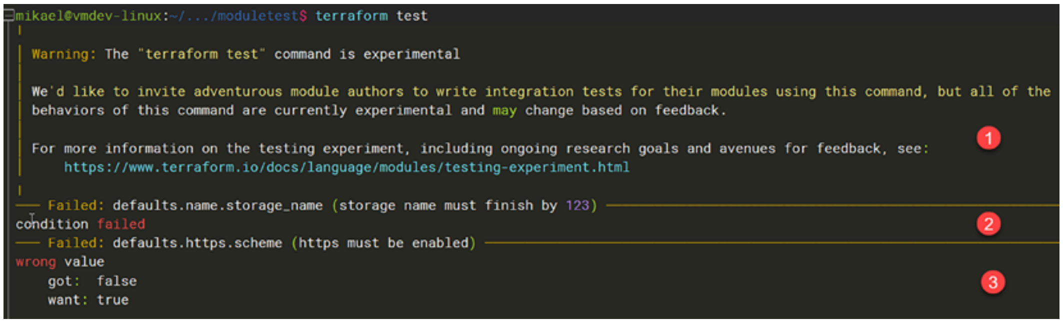
In *step 3*, in this file, we add the Terraform resource `test_assertion`, which tests the storage account name requirement by checking that the `storage_name` module output finishes with `123` using a regular expression.

Then, in *step 4*, inside the `moduletest` folder, we run the `terraform test` command.

This command performs the following actions:

1. Provision the resources for the module by running the `terraform init`, `plan`, and `apply` commands.
2. Execute the tests.
3. Destroy the provisioned resources by running the `terraform destroy` command.
4. Display the test results.

The following image shows the output of this command.



```
mikael@vmdev-linux:~/.../moduletest$ terraform test
 |
 | Warning: The "terraform test" command is experimental
 |
 | We'd like to invite adventurous module authors to write integration tests for their modules using this command, but all of the
 | behaviors of this command are currently experimental and may change based on feedback.
 |
 | For more information on the testing experiment, including ongoing research goals and avenues for feedback, see:
 |     https://www.terraform.io/docs/language/modules/testing-experiment.html
 |
 ── Failed: defaults.name.storage_name (storage name must finish by 123) ──────────────
condition failed
── Failed: defaults.https.scheme (https must be enabled) ──────────────────────────
wrong value
    got:  false
    want: true
```

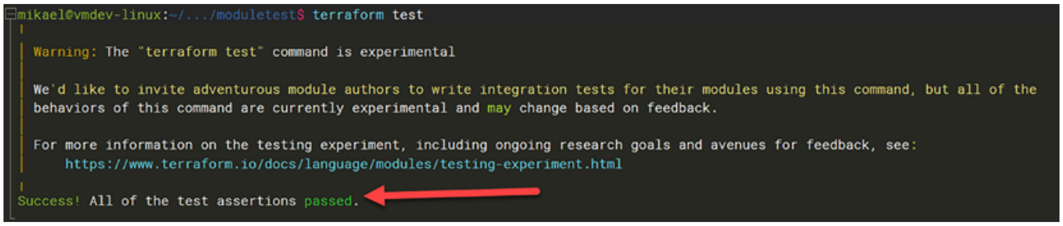
Figure 11.15: The terraform test command execution with an error

In this output, we can see the following elements:

- Warning about the experimental feature (bullet 1)
- The test on the storage that is failing because the name doesn't end with 123 (bullet 2)
- The test on HTTPS that is failing because the code is configured for HTTP (bullet 3)

Now, in *step 5*, we will fix the Terraform configuration of the module to be compliant with the specifications (HTTPS and the name).

Finally, to check the configuration, we rerun the `terraform test` command; the following image shows the output of this command:



```
mikael@vmdev-linux:~/.../moduletest$ terraform test
│
│ Warning: The "terraform test" command is experimental
│
│ We'd like to invite adventurous module authors to write integration tests for their modules using this command, but all of the
│ behaviors of this command are currently experimental and may change based on feedback.
│
│ For more information on the testing experiment, including ongoing research goals and avenues for feedback, see:
│     https://www.terraform.io/docs/language/modules/testing-experiment.html
│
Success! All of the test assertions passed.
```

Figure 11.16: Successful execution of the Terraform test command

We can see that all tests are passing successfully.

There's more...

As we have said a lot, this command is still in the experimental stage and still has some shortcomings in my opinion.

For example, when we execute this command, we do not see any log on what the command operates; only the result of the tests is displayed at the end – for example, the `init`, `plan`, `apply`, and `destroy` command executions, which are done on the module configuration and are not displayed in the logs of the command. And if the `apply` command fails, not only does the command not display it but also mentions that the test results are successful.

Additionally, the `terraform test` command has some optional arguments as follows:

- `compact-warning` to compact the warning message
- `junit-xml` to export the test results in the Junit XML format, to publish this XML file in a CI/CD system

To get more details about the command options, run the `terraform test -help` command.

See also

- For more information about Terraform testing, read the article here: `https://www.hashicorp.com/blog/testing-hashicorp-terraform`
- The documentation for this experimental testing feature is available here: `https://developer.hashicorp.com/terraform/language/modules/testing-experiment`
- The documentation for the `terraform test` command is available here: `https://developer.hashicorp.com/terraform/cli/commands/test`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

12

Deep-Diving into Terraform

In this book, we started with recipes for Terraform that concern its installation, writing the Terraform configuration, as well as examining the use of the Terraform CLI commands. Then, we studied sharing the Terraform configuration using modules. Finally, we focused on the use of Terraform to build Azure, GCP, AWS, and Kubernetes infrastructures.

Now, in this chapter, we will discuss recipes that allow us to go further in our usage of Terraform. We will learn how to use the templates in Terraform to generate an Ansible inventory file. We will discuss how to prevent the destruction of resources, how to implement a zero-downtime deployment with Terraform, and how to detect the deletion of resources when Terraform applies changes.

Then, we will discuss the use of Terragrunt to manage the workspace dependency and its use as a wrapper for the Terraform CLI.

Finally, we will talk about the use of Git hooks to check in the Terraform configuration before committing the configuration and Rover to visualize the Terraform resource dependencies, and get an overview of the Terraform CDK for abstracting Terraform configuration with a higher-level language like TypeScript.

In this chapter, we will cover the following recipes:

- Preventing resources from being destroyed
- Ignoring manual changes
- Using Terraform’s templating feature
- Zero-downtime deployment with Terraform
- Managing Terraform configuration dependencies using Terragrunt
- Using Terragrunt as a wrapper for Terraform

- Generating a self-signed certificate using Terraform
- Checking the configuration before committing code using Git hooks
- Visualizing Terraform resource dependencies with Rover
- Using the Terraform CDK for developers

Technical requirements

For the recipes in this chapter, we will need the following tools:

- Terragrunt, whose documentation is available at https://terragrunt.gruntwork.io/.
- We will also use the **jq** utility to parse JSON. You can download it from `https://stedolan.github.io/jq/`.
- We will use Node.js to install it, refer to the documentation here: `https://nodejs.org/en`.

The complete source code for this chapter is available here:

`https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12`

Preventing resources from being destroyed

The use of **Infrastructure as Code** (**IaC**) requires attention in some cases. Indeed, when the Terraform execution is integrated into a CI/CD pipeline and the plan is overlooked, resources containing important data can be deleted. This can be done either by changing a property of a Terraform resource, which requires the deletion and recreation of this resource, or by executing the `terraform destroy` command.

Fortunately, Terraform includes a configuration in its language that prevents the destruction of sensitive resources.

In this recipe, we will see how to prevent the destruction of resources that are managed in a Terraform configuration.

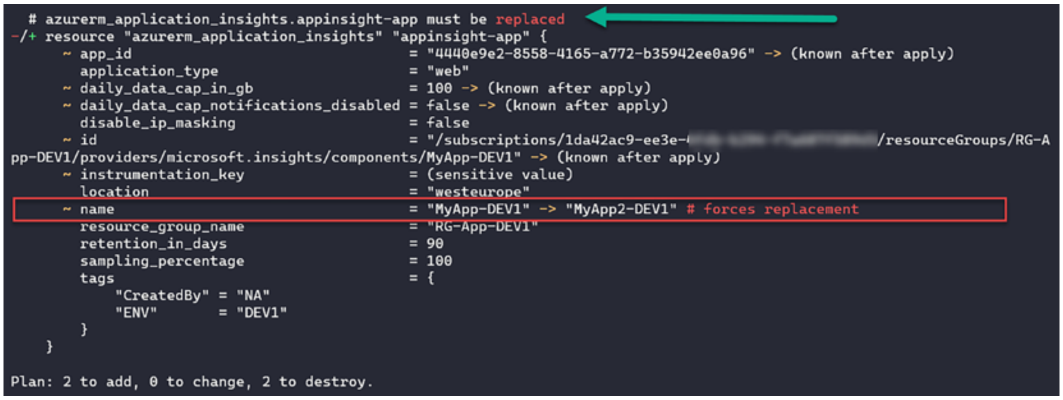
Getting ready

For this recipe, we will use a Terraform configuration to manage the following resources in Azure:

- An Azure Resource Group
- An Azure App Service plan
- An Azure App Service (web app) instance
- An Azure Application Insights instance

We often encounter in company projects that resources contain valuable data; in our example, we have the Application Insights instance containing the logs and metrics of our application, which is in the Azure Web App. We don't want the Application Insights instance to be deleted automatically and for us to lose its data.

Let's take as a scenario a company that has decided to change the nomenclature of its resources, and we need to update the Terraform configuration with the new nomenclature. When running Terraform, we would get the following result from the `terraform plan` command:



```
  # azurerm_application_insights.appinsight-app must be replaced
-/+ resource "azurerm_application_insights" "appinsight-app" {
      ~ app_id                                = "4440e9e2-8558-4165-a772-b35942ee0a96" -> (known after apply)
        application_type                      = "web"
      ~ daily_data_cap_in_gb                  = 100 -> (known after apply)
      ~ daily_data_cap_notifications_disabled = false -> (known after apply)
        disable_ip_masking                    = false
      ~ id                                    = "/subscriptions/1da42ac9-ee3e-[redacted]/resourceGroups/RG-A
pp-DEV1/providers/microsoft.insights/components/MyApp-DEV1" -> (known after apply)
      ~ instrumentation_key                   = (sensitive value)
        location                              = "westeurope"
      ~ name                                  = "MyApp-DEV1" -> "MyApp2-DEV1" # forces replacement
        resource_group_name                   = "RG-App-DEV1"
        retention_in_days                     = 90
        sampling_percentage                   = 100
        tags                                  = {
            "CreatedBy" = "NA"
            "ENV"       = "DEV1"
        }
    }

Plan: 2 to add, 0 to change, 2 to destroy.
```

Figure 12.1: Terraform recreates a resource

As you can see, the name change requires the deletion of the Application Insights instance, which contains important log metrics.

The purpose of this recipe is to change the Terraform configuration so that the Application Insights resource is never deleted.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/preventdestroy`.

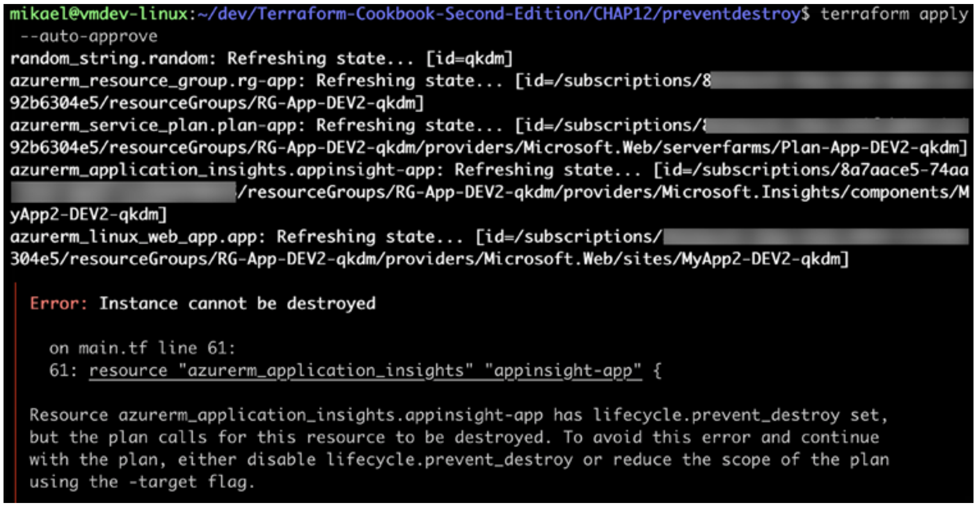
How to do it...

To prevent the deletion of a resource by Terraform, perform the following steps:

1. Inside the Terraform configuration of the Application Insights resource, add the following `lifecycle` block:

```
resource "azurerm_application_insights" "appinsight-app" {
...
  lifecycle {
      prevent_destroy = true
    }
  }
```

2. In `variables.tf`, change the default value of the `app_name` variable to another name for the Application Insights instance, such as `MyApp2-DEV1`.
3. Execute the Terraform CLI workflow – the result is shown in the following screenshot:



```
mikael@vmdev-linux:~/dev/Terraform-Cookbook-Second-Edition/CHAP12/preventdestroy$ terraform apply
 --auto-approve
random_string.random: Refreshing state... [id=qkdm]
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/8[redacted]
92b6304e5/resourceGroups/RG-App-DEV2-qkdm]
azurerm_service_plan.plan-app: Refreshing state... [id=/subscriptions/[redacted]
92b6304e5/resourceGroups/RG-App-DEV2-qkdm/providers/Microsoft.Web/serverfarms/Plan-App-DEV2-qkdm]
azurerm_application_insights.appinsight-app: Refreshing state... [id=/subscriptions/8a7aace5-74aa
[redacted]/resourceGroups/RG-App-DEV2-qkdm/providers/Microsoft.Insights/components/M
yApp2-DEV2-qkdm]
azurerm_linux_web_app.app: Refreshing state... [id=/subscriptions/[redacted]
304e5/resourceGroups/RG-App-DEV2-qkdm/providers/Microsoft.Web/sites/MyApp2-DEV2-qkdm]

│ Error: Instance cannot be destroyed
│
│   on main.tf line 61:
│   61: resource "azurerm_application_insights" "appinsight-app" {
│
│ Resource azurerm_application_insights.appinsight-app has lifecycle.prevent_destroy set,
│ but the plan calls for this resource to be destroyed. To avoid this error and continue
│ with the plan, either disable lifecycle.prevent_destroy or reduce the scope of the plan
│ using the -target flag.
```

Figure 12.2: Terraform prevent_destroy resource

How it works...

In this recipe, we have added the Terraform `lifecycle` block, which contains the properties that allow interaction with resource management. Here, in our case, we used the `prevent_destroy` property, which, as its name indicates, prevents the destruction of the specified resource.

There's more...

As we have discussed, the `prevent_destroy` property allows you to prohibit the deletion of resources.

Note that in our example with Azure, this property does not prohibit the deletion of resources via the Azure portal or the Azure CLI.

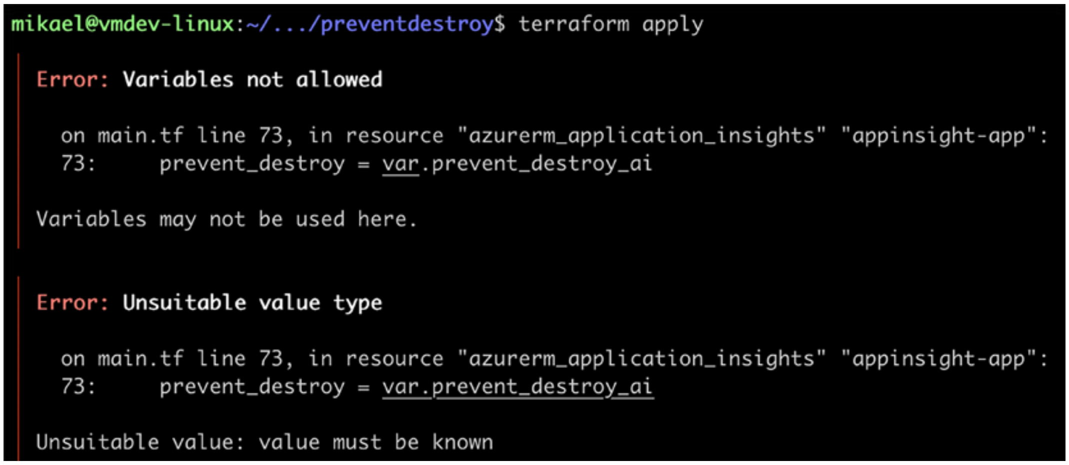
However, it should be noted that if a resource in the Terraform configuration contains this property, and this property must be deleted when executing the `terraform apply` command, then this `prevent_destroy` property prevents the application from making changes to all the resources described in the Terraform configuration. This blocks us from applying changes to resources if those changes include the destruction of that resource. This is one of the reasons why it is recommended to break up the Terraform configuration, putting the configuration of the sensitive resources that mustn't be destroyed into one folder (and thus a separate Terraform state file), and the other resources into another folder. This way, we can apply changes to the resources without being blocked by our resource destruction prevention settings.

Here, I'm writing about separating the Terraform configuration and the terraform state, but it's also necessary to separate the workflows in the CI/CD pipeline, with one pipeline that applies the changes and another that destroys the resources.

In addition, mostly to prevent human mistakes, it isn't possible to add variables to the values of the properties of the `lifecycle` block. If you wanted to make the value of this property dynamic, you might be tempted to use a bool-type variable, such as in the following code:

```
lifecycle {
 prevent_destroy = var.prevent_destroy_ai
}
```

However, when executing the `terraform apply` command, the following error occurs:



```
mikael@vmdev-linux:~/.../preventdestroy$ terraform apply

 Error: Variables not allowed

   on main.tf line 73, in resource "azurerm_application_insights" "appinsight-app":
   73:     prevent_destroy = var.prevent_destroy_ai

 Variables may not be used here.


 Error: Unsuitable value type

   on main.tf line 73, in resource "azurerm_application_insights" "appinsight-app":
   73:     prevent_destroy = var.prevent_destroy_ai

 Unsuitable value: value must be known
```

Figure 12.3: Terraform prevent_destroy is not allowed as a variable

These errors indicate that a variable is not allowed in the `lifecycle` block, so you must keep true/false values in the code.

See also

- Documentation on the `prevent_destroy` property is available at `https://www.terraform.io/docs/configuration/resources.html#prevent_destroy`.
- An interesting article on the HashiCorp blog about drift management can be found at `https://www.hashicorp.com/blog/detecting-and-managing-drift-with-terraform/`.
- Read this article from HashiCorp about feature toggles, blue-green deployments, and canary testing using Terraform, available at `https://www.hashicorp.com/blog/terraform-feature-toggles-blue-green-deployments-canary-test/`.

Ignoring manual changes

In the previous recipe, we learned how to prevent resources from being deleted when using Terraform by using the `prevent_destroy` property.

In some situations, which need to be measured, we need to modify the resource properties manually, i.e., without having to modify the Terraform configuration. And as we know, if we modify a resource outside the Terraform configuration, the next time we apply Terraform, the changes we made manually will be overwritten by the Terraform configuration. When Terraform is applied, it performs a refresh step that reads the current state of the infrastructure and compares it to the desired state described in the configuration files. If Terraform detects any changes between the two, it will attempt to modify the infrastructure to match the desired state.

This is the purpose of IaC, to have the code be the source of truth for the state of the infrastructure.

In this recipe, we will learn how to update the Terraform configuration to allow us to update resources manually and not overwrite the resource in the next execution of the `terraform apply` command.

Let’s get started!

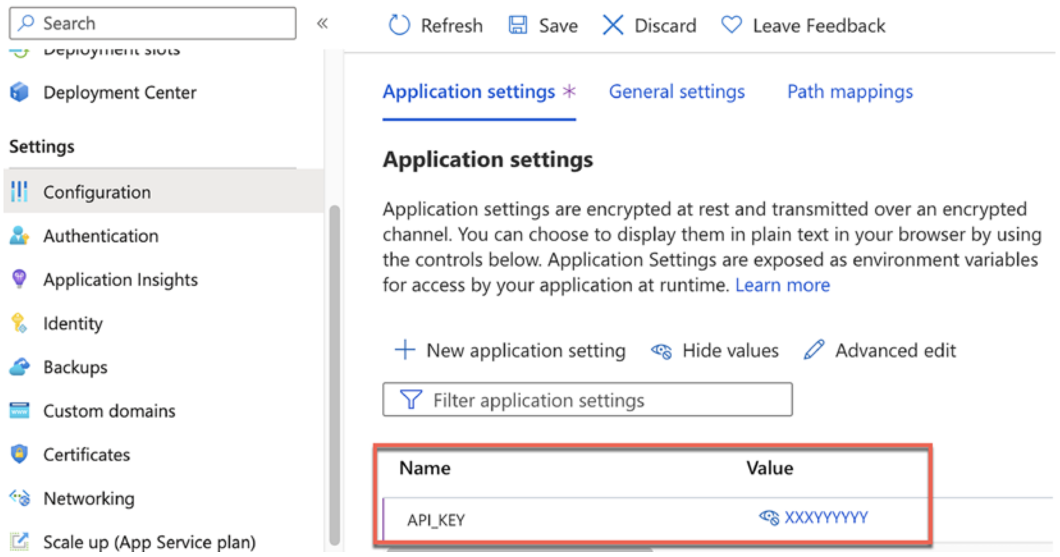
Getting ready

To illustrate this recipe, we will provision a basic Azure resource composed of an Azure Resource Group, an Azure Service Plan, and an Azure App Service using a Terraform configuration. (The source code for this Terraform configuration is included in the source code for this recipe.)

To provision the Azure resources, go to the `CHAP12/ignorechanges/` folder found in this book's GitHub repository. Execute the Terraform workflow running the `init`, `plan` and `apply` commands. Doing so will create four resources

In this scenario, after provisioning the resources, developers want to add or update the application settings of the provisioned App Service directly via the Azure portal.

The following image shows **Application settings** in the Azure portal edited by users:



Application settings

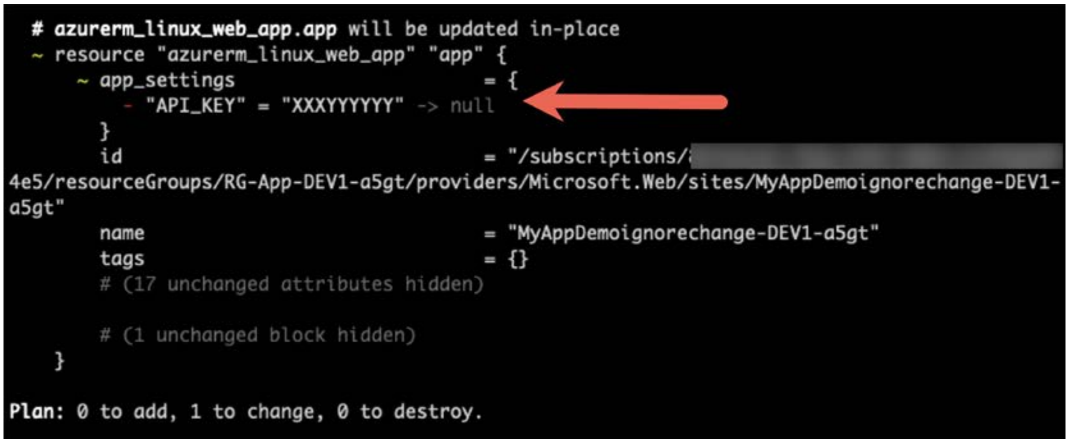
Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose to display them in plain text in your browser by using the controls below. Application Settings are exposed as environment variables for access by your application at runtime. Learn more

Name	Value
API_KEY	XXXYYYYYY

Figure 12.4: Application settings in the Azure portal

Users add the application settings **API_KEY** directly via the Azure portal.

The problem is when we run `terraform plan` on the configuration, we get the following result:



```
  # azurerm_linux_web_app.app will be updated in-place
  ~ resource "azurerm_linux_web_app" "app" {
      ~ app_settings                      = {
          - "API_KEY" = "XXXYYYYYY" -> null
        }
        id                                = "/subscriptions/[redacted]
4e5/resourceGroups/RG-App-DEV1-a5gt/providers/Microsoft.Web/sites/MyAppDemoignorechange-DEV1-
a5gt"
        name                              = "MyAppDemoignorechange-DEV1-a5gt"
        tags                              = {}
        # (17 unchanged attributes hidden)

        # (1 unchanged block hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.
```

Figure 12.5: Terraform changes the value

We can see that the configuration will overwrite the user's changes.

The goal of this recipe is to allow us to update the application settings manually without impacting the future execution of the `terraform apply` command for this Terraform configuration.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/ignorechanges`.

How to do it…

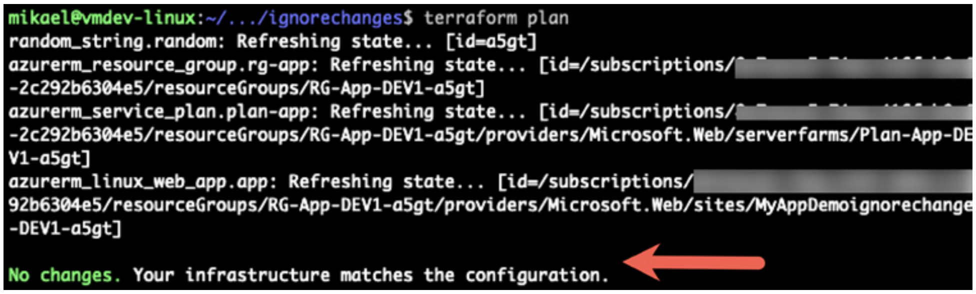
To allow manual changes, perform the following steps:

1. Inside the configuration of the Azure Web App, add the meta-argument `ignore_changes` in the `lifecycle` block with the following configuration:

```
resource "azurerm_linux_web_app" "app" {
….

  app_settings = {}

  lifecycle {
    ignore_changes = [
      app_settings
    ]
  }
}
```

2. Run the commands `terraform init` and `terraform plan` to check that the App Service instance will not be changed.

   The following image shows the output of `terraform plan`:



```
mikael@vmdev-linux:~/.../ignorechanges$ terraform plan
random_string.random: Refreshing state... [id=a5gt]
azurerm_resource_group.rg-app: Refreshing state... [id=/subscriptions/[redacted]
-2c292b6304e5/resourceGroups/RG-App-DEV1-a5gt]
azurerm_service_plan.plan-app: Refreshing state... [id=/subscriptions/[redacted]
-2c292b6304e5/resourceGroups/RG-App-DEV1-a5gt/providers/Microsoft.Web/serverfarms/Plan-App-DE
V1-a5gt]
azurerm_linux_web_app.app: Refreshing state... [id=/subscriptions/[redacted]
92b6304e5/resourceGroups/RG-App-DEV1-a5gt/providers/Microsoft.Web/sites/MyAppDemoignorechange
-DEV1-a5gt]

No changes. Your infrastructure matches the configuration.
```

Figure 12.6: Terraform manually changes with ignore_change

   No changes will be applied by Terraform.

How it works…

In this recipe, we add the `ignore_changes` meta-argument inside the `lifecycle` block to ignore changes to application settings.

When you use the `ignore_changes` meta-argument in Terraform, it tells Terraform to skip comparing certain resource attributes between the desired state and the current state during `terraform apply`.

There's more…

When Terraform applies a configuration, it compares the desired state of the infrastructure with the current state of the infrastructure. It does this by examining the configuration files and also by querying the API of the cloud provider (such as Azure, AWS, or GCP) to get the current state of the resources. Then, it determines what changes need to be made to bring the desired state and the current state into alignment.

However, when you specify an attribute in the `ignore_changes` block, Terraform will ignore any changes to that attribute during this comparison process. This means that if the current state of the infrastructure has a different value for the ignored attribute from what is specified in the configuration file, Terraform will not attempt to modify the infrastructure to match the configuration file. Instead, it will simply leave the current value in place.

It's important to note that `ignore_changes` should be used with care, as it can potentially lead to configuration drift if the current state of the infrastructure diverges significantly from the desired state specified in the configuration file. Therefore, you should only use `ignore_changes` if you are sure that it's safe to do so and you fully understand the consequences.

See also

- The documentation for the `ignore_changes` meta-argument is available here: `https://developer.hashicorp.com/terraform/language/meta-arguments/lifecycle#ignore_changes`.

Using Terraform's templating feature

Terraform is a very good **IaC** tool that allows us to build complex infrastructure with code. One of Terraform's features is the ability to generate text or files based on templates. To illustrate this feature, let's take a look at a use case I came upon in one of our companies. Of course, there are plenty of use cases for templating with Terraform.

The scenario I'd like to illustrate is the possibility of generating an Ansible inventory file containing the list of host VMs to be configured (using Ansible playbooks) from Terraform configuration, which will have previously provisioned these VMs.

As we studied in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*, concerning the construction of virtual machines, on all cloud providers, the common objective of Terraform is to build a VM without configuring it, which includes the installation of its middleware and its administration.

Ansible (`https://www.ansible.com/`), is very popular in the open-source world (much like Chef and Puppet). The installation documentation is available here: `https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html`.

One of the advantages of Ansible is that it's agentless, which means you don't need to install an agent on the VMs you want to configure. Thus, to know which VMs to configure, Ansible uses a file called *inventory*, which contains the list of VMs that need configuring.

In this recipe, we will learn how to generate this *inventory* file using Terraform's templating features.

Getting ready

The purpose of this recipe is not to discuss the installation and use of Ansible but just the automatic creation of its *inventory* file.

To learn more about Ansible, I invite you to read *Chapter 3*, *Using Ansible for Configuring IaaS Infrastructure*, of my book entitled *Learning DevOps*, also available from Packt at `https://www.packtpub.com/eu/cloud-networking/learning-devops`.

The starting point of our recipe is to use Terraform to create VMs in Azure whose private IP addresses are not known before they are created. In this Terraform configuration of VMs, we use the configuration we already studied in the *Provisioning and configuring an Azure VM with Terraform* recipe of *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*. So, to keep it simple, we use the Terraform modules published in the public registry with the following Terraform configuration:

1. Instantiate a `vmhosts` variable that specifies the hostname of the VM we want to create:

```
variable "vmhosts" {
  type    = list(string)
  default = ["vmwebdemo1", "vmwebdemo2"]
}
```

2. Then, use the `network` module and compute from the public registry to create the VM inside the network:

```
module "network" {
  source = "Azure/network/azurerm"
  resource_group_name = "rg-demoinventory"
  subnet_prefixes = ["10.0.2.0/24"]
  subnet_names = ["subnet1"]
  use_foreach = true
}

module "linuxservers" {
  source = "Azure/compute/azurerm"
  resource_group_name = "vmwebdemo-${random_string.random.result}"
  vm_os_simple = "UbuntuServer"
  nb_instances = 2
  nb_public_ip = 2
  vm_hostname = "azurerm_resource_group.rg.name"
  vnet_subnet_id = azurerm_subnet.snet1.id
}
```

In the preceding Terraform configuration, we create a virtual network, a subnet, and two Linux VMs, which will have private IP addresses.

The goal of this recipe is to generate an `inventory` text file, in the same Terraform configuration, which will contain the list of hosts (along with their IP addresses) that have been created by Terraform. This inventory file will be in the following form:

```
[vm-web]
<host1> ansible_host=1<ip 1>
<host2> ansible_host=<ip 2>
```

The complete source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/ansible-inventory`.

How to do it...

To generate the Ansible inventory file with Terraform, perform the following steps:

1. Inside the folder containing the Terraform configuration, we create a new file called `template-inventory.tpl` with the following content:

```
[vm-web]
%{ for host, ip in vm_dnshost ~}
${host} ansible_host=${ip}
%{ endfor ~}
```

2. Then, in the `main.tf` file of the Terraform configuration that creates a VM, we add the following code to generate the `inventory` file:

```
resource "local_file" "inventory" {
  filename = "inventory"
  content = templatefile("template-inventory.tpl",
    {
         vm_dnshost = zipmap(var.vmhosts,module.linuxservers.
network_interface_private_ip)
    })
}
```

3. Finally, to create the VMs and generate the `inventory` file, we run the basic Terraform `init`, `plan`, and `apply` workflow commands.

How it works...

We first create a `template-inventory.tpl` file, which uses Terraform's template format. In this file, we use a `for` loop with the syntax `%{ for` `host`, `ip` `in` `vm_dnshost` `~}`, which allows us to loop the elements of the `vm_dnshost` variable. For each VM in this loop, we use the following syntax:

```
${host} ansible_host=${ip}
```

We end the loop with the `%{ endfor ~}` syntax.

For more details on this templating format, read the documentation at `https://www.terraform.io/docs/configuration/expressions.html#string-templates`.

Then, in *Step 2*, to the Terraform configuration, we add a `local_file` resource (which we already studied in the *Manipulating local files with Terraform* recipe in *Chapter 2*, *Writing Terraform Configurations*) in which we fill in the following properties:

- `filename`: This contains `inventory` as its value, which is the name of the file that will be generated.

  In this recipe, the file will be generated inside the directory that currently contains this Terraform configuration. You are free to enter another folder for generation and storage.

- `content`: This contains the elements that will fill this file. Here, we use the `templatefile` function, passing the following as parameters:
  - The name of the template file, `template-inventory.tpl`, that we created in *Step 1*.
  - The `vm_dnshost` variable, which will fill the content of the template file. We use the built-in Terraform `zipmap` function, which allows us to build a map from two lists, one being the keys list and the other the values list.

  Documentation on the `zipmap` function is available at `https://www.terraform.io/docs/configuration/functions/zipmap.html`.

Finally, in the last step, we execute the commands of the Terraform workflow, and at the end of its execution, we can see that the `inventory` file has indeed been generated with the following content:

```
[vm-web]
vmwebdemo1 ansible_host=10.0.2.5
vmwebdemo2 ansible_host=10.0.2.4
```

Now, all new VMs added to this Terraform configuration will be added to this Ansible inventory.

There's more...

The primary objective of this recipe is to show the use of templates with Terraform, which we applied to an Ansible inventory. There are several other use cases for these templates, such as using the `cloud-init` file to configure a VM, which is explained in the article at `https://grantorchard.com/dynamic-cloudinit-content-with-terraform-file-templates/`.

See also

- The documentation on the Terraform `templatefile` function is available at `https://www.terraform.io/docs/configuration/functions/templatefile.html`.
- The documentation on the `local_file` resource of the `local` provider is available at `https://registry.terraform.io/providers/hashicorp/local/latest/docs/resources/file`.
- A list of books on Ansible from Packt is available at `https://subscription.packtpub.com/search?query=ansible`.
- Here is a list of web articles that deal with the same subject of Ansible inventories generated by Terraform by proposing different solutions:
    - `https://hooks.technology/posts/ansible-terraform/`
    - `http://web.archive.org/web/20210921222018/https://www.linkbynet.com/produce-an-ansible-inventory-with-terraform`
    - `https://gist.github.com/hectorcanto/71f732dc02541e265888e924047d47ed`
    - `https://stackoverflow.com/questions/45489534/best-way-currently-to-create-an-ansible-inventory-from-terraform`

Zero-downtime deployment with Terraform

As discussed in the previous recipe, changing certain properties of resources described in the Terraform configuration can lead to their destruction and subsequent recreation. Resources are destroyed and recreated in the order in which they depend on each other (if they do). The default behavior when recreating a resource involves first destroying the old one and then creating a new one, and for certain resources in a production context, during this time period, this will lead to downtime, that is, a service interruption. This downtime will be greater or smaller depending on the type of resources that have to be destroyed and then recreated.

For example, in Azure, a VM takes much longer to destroy and rebuild than a Web App or a **Network Security Group** (**NSG**) rule.

In Terraform, there is a mechanism that allows for zero downtime and therefore avoids this service interruption when deleting a resource.

In this recipe, we will study how to implement zero downtime on a resource described in a Terraform configuration.

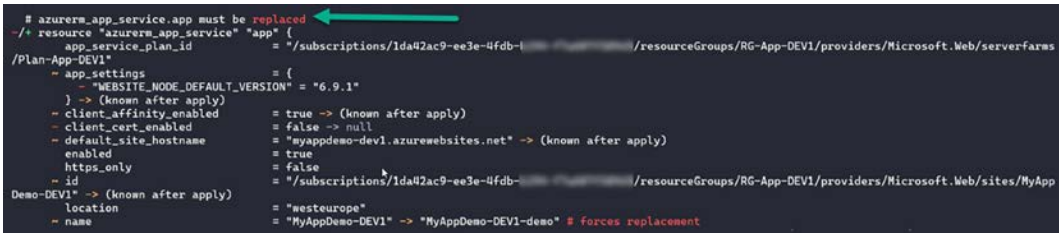
Getting ready

For this recipe, we will use the Terraform configuration available here – `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/zerodowntime` – which allows us to provision the following resources in Azure:

- An Azure Resource Group
- An Azure Service Plan
- An Azure App Service (web app) instance
- An Application Insights instance

In addition, this Terraform configuration has already been applied to the Azure cloud.

For our use case, let's assume that a company has decided to change the resource name and that we need to update the Terraform configuration with the new name. When running Terraform, the following result would be obtained by the `terraform plan` command:



```
  # azurerm_app_service.app must be replaced
-/+ resource "azurerm_app_service" "app" {
        app_service_plan_id            = "/subscriptions/1da42ac9-ee3e-4fdb-[redacted]/resourceGroups/RG-App-DEV1/providers/Microsoft.Web/serverfarms
/Plan-App-DEV1"
      ~ app_settings                   = {
          - "WEBSITE_NODE_DEFAULT_VERSION" = "6.9.1"
        } -> (known after apply)
      ~ client_affinity_enabled        = true -> (known after apply)
      - client_cert_enabled            = false -> null
      ~ default_site_hostname          = "myappdemo-dev1.azurewebsites.net" -> (known after apply)
        enabled                        = true
        https_only                     = false
      ~ id                             = "/subscriptions/1da42ac9-ee3e-4fdb-[redacted]/resourceGroups/RG-App-DEV1/providers/Microsoft.Web/sites/MyApp
Demo-DEV1" -> (known after apply)
        location                       = "westeurope"
      ~ name                           = "MyAppDemo-DEV1" -> "MyAppDemo-DEV1-demo" # forces replacement
```

Figure 12.7: Terraform destroying a resource

As you can see, the name change requires the deletion of the Azure Web App that hosts our web application. This deletion would result in the application not being accessible for a small amount of time while it is recreated. The purpose of this recipe is to modify the Terraform configuration so that even when the App Service resource is deleted, the web application will still be available.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/zerodowntime`.

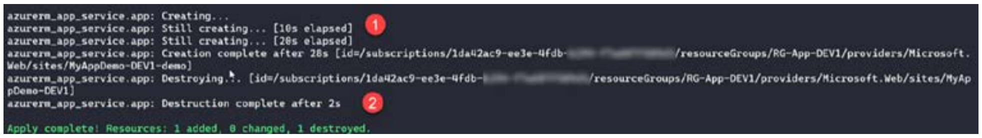
How to do it...

To provide zero downtime in a Terraform configuration, perform the following steps:

1. In the Terraform configuration, inside the `azurerm_app_service` resource, add the `lifecycle` block, as shown in the following code:

```
resource "azurerm_app_service" "app" {
  name                  = "${var.app_name}-${var.environement}"
...
  lifecycle {
    create_before_destroy = true
  }
}
```

2. Change the `name` property of the App Service to apply the new nomenclature.
3. Execute the Terraform CLI workflow and the `terraform apply` result will be shown, as in the following screenshot:



```
azurerm_app_service.app: Creating...
azurerm_app_service.app: Still creating... [10s elapsed]
azurerm_app_service.app: Still creating... [20s elapsed]
azurerm_app_service.app: Creation complete after 28s [id=/subscriptions/1da42ac9-ee3e-4fdb-[redacted]/resourceGroups/RG-App-DEV1/providers/Microsoft.
Web/sites/MyAppDemo-DEV1-demo]
azurerm_app_service.app: Destroying... [id=/subscriptions/1da42ac9-ee3e-4fdb-[redacted]/resourceGroups/RG-App-DEV1/providers/Microsoft.Web/sites/MyAp
pDemo-DEV1]
azurerm_app_service.app: Destruction complete after 2s

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.
```

Figure 12.8: Terraform creating a resource before destroying it

How it works...

In *Step 2*, we added the `lifecycle` block to the `azurerm_app_service` resource. In this block, we added the `create_before_destroy` property with its value set to `true`. This property makes the regeneration of a resource possible in the event of destruction by indicating to Terraform to first recreate the resource, and only then to delete the original resource.

There's more...

As we've seen, by using this property, there is no more interruption of service. As long as the new resource is not created, the old one is not deleted, and the application continues to be online.

However, before using `create_before_destroy`, there are some things to consider, as follows:

The `create_before_destroy` property only works when a configuration change requires the deletion and then regeneration of resources. It only works when executing the `terraform apply` command; it does not work when executing the `terraform destroy` command.

You must be careful that the names of the resources that will be created have different names from the ones that will be destroyed afterward. Otherwise, if the names are identical, the resource may not be created.

Moreover, this zero-downtime technique is only really effective if the resource that will be impacted is fully operational at the end of its creation. For example, let's take the case of a VM: although Terraform can quickly create it, it's still fully functional after being recreated (the installation of the middleware and deployment of the application). All this configuration can generate downtime, and in order to be efficient in this case, I advise you to use Packer from HashiCorp (`https://www.packer.io/`), which allows you to create images of VMs that are already fully configured.

> To implement zero downtime in Azure with Packer and Terraform, read the tutorial at `https://docs.microsoft.com/en-us/azure/developer/terraform/create-vm-scaleset-network-disks-using-packer-hcl`.

Finally, we have seen in this recipe how to implement zero-downtime deployments with Terraform, but according to your provider, there are most likely other practices that are native to them. For example, we can also use load balancers, and for an App Service instance on Azure, we can use slots, as explained in the documentation at `https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots`.

See also

- Read the HashiCorp blog post about the `create_before_destroy` property at `https://www.hashicorp.com/blog/zero-downtime-updates-with-terraform/`.
- A good article on zero downtime can be found at `https://dzone.com/articles/zero-downtime-deployment`.

Managing Terraform configuration dependencies using Terragrunt

In several recipes in this book, we have discussed the organization of the files that contain the Terraform configuration. We examined this more specifically in the *Provisioning infrastructure in multiple environments* recipe in *Chapter 2*, *Writing Terraform Configurations*, which outlines several architecture solutions.

One of the best practices regarding the structure of the configuration is to separate the Terraform configuration into infrastructure and application components, as explained in the article at `https://www.cloudreach.com/en/resources/blog/how-to-simplify-your-terraform-code-structure/`. The challenge with a structure split into several configurations is the maintenance of dependencies and run schedules between these components.

Among all the third-party tools that revolve around Terraform, there is **Terragrunt** (`https://terragrunt.gruntwork.io/`), developed by Gruntwork. **Terragrunt** is open-source and offers a lot of additional functionality for the organization and execution of Terraform configurations.

In this recipe, we will learn how you can use **Terragrunt** to manage the dependencies of different Terraform workspace dependencies.

Getting ready

For this recipe, we must have previously installed the **Terragrunt** binary on our workstations by following the instructions at `https://terragrunt.gruntwork.io/docs/getting-started/install/#install-terragrunt`.

Before installing Terragrunt, check the version compatibility with Terraform by using the following link: `https://terragrunt.gruntwork.io/docs/getting-started/supported-terraform-versions/`

In this recipe, we will build an infrastructure consisting of the following elements:

- An Azure Resource Group
- An Azure network with a virtual network and a subnet
- An Azure VM

The architecture of the folders containing this Terraform configuration is as follows:

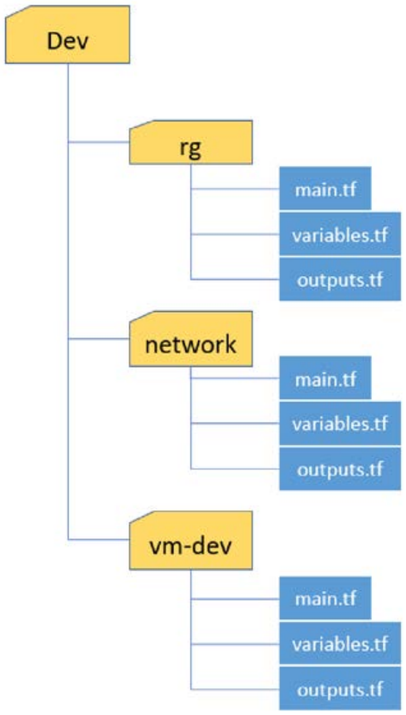


Figure 12.9: Terraform folder configuration structure

The problem with this architecture is the dependency between configurations and the fact that they must be executed in a specific order. Indeed, to apply the network, the Resource Group must be applied first, and it's the same for the VM: the network must be created beforehand. With Terraform, in the case of several changes, the Terraform workflow must be executed several times and in the correct order for each of those configurations.

The purpose of this recipe is not to explain all the functionalities of **Terragrunt** in detail, but to demonstrate one of its features, which is to simplify the execution of Terraform when the Terraform configuration is separated into several folders that are linked by dependencies.

The source code for this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/demogrunt/`.

How to do it...

Perform the following steps to use **Terragrunt** with Terraform dependencies:

1. To add the dependency between the `network` and `rg` configurations, inside the `network` folder, add a new file called `terragrunt.hcl` with the following content:

   ```
   dependencies {
     paths = ["../rg"]
   }
   ```

2. Inside the `vm-web` folder, add a new file called `terragrunt.hcl` with the following content:

   ```
   dependencies {
     paths = ["../network"]
   }
   ```

3. In a terminal, inside the `dev` folder, run the following `terragrunt` commands to create the Resource Group, the Azure Virtual Network, and the VM:

   ```
   terragrunt run-all init
   terragrunt run-all apply --terragrunt-non-interactive
   ```

How it works...

The `terragrunt.hcl` file we added contains the configuration for **Terragrunt**.

Here, in the configuration we wrote in *Step 2*, we indicated a dependency between the network configuration and the Resource Group configuration (because the Resource Group must be created before executing the network configuration) and the dependency between the VM and the network.

In *Step 3*, we then executed the **Terragrunt** commands (`terragrunt run-all init` and `terragrunt run-all apply`) and when executing them, thanks to the configuration we wrote, **Terragrunt** will first apply Terraform in the correct order of dependency that we configured in the file `terragrunt.hcl`.

So the application order is to start with the Resource Group, then move to the network automatically, and finally handle the VM. This is without having to apply the Terraform workflow several times to several Terraform configurations and in the right order.

There's more...

As you saw in this recipe, we did not execute the `terragrunt run-all plan` command because, in the Terraform configuration, we used resources of type data, which reference resources that must be created in the dependency configuration (which must be applied before the current configuration). And so the execution of the `terraform plan` command doesn't work until the Terraform dependency configuration has been applied.

In this recipe, we studied how to improve the dependency between the Terraform configuration using Terragrunt and its configuration. We can go even further with this improvement, externalizing the configuration (which is redundant between each environment) by reading the documentation available at `https://terragrunt.gruntwork.io/docs/features/execute-terraform-commands-on-multiple-modules-at-once/`.

However, since **Terragrunt** runs the Terraform binary that is installed on your local computer, you should make sure to install a version of **Terragrunt** that is compatible with the version of the Terraform binary installed.

In the next recipe, we will complete the configuration of **Terragrunt** to be able to use it as a wrapper for Terraform by simplifying the Terraform command lines.

See also

- Detailed documentation for Terragrunt is available at .
- The source code for Terragrunt is available on GitHub at `https://github.com/gruntwork-io/terragrunt`.
- A useful blog article on the architecture of the Terraform configuration can be found at `https://www.hashicorp.com/blog/structuring-hashicorp-terraform-configuration-for-production/`.

Using Terragrunt as a wrapper for Terraform

In the many years that I have worked and supported customers with Terraform, a recurring problem has prevented users from making full use of Terraform's functionality. What I have noticed is that these users do not encounter any problems with the language and the writing of the provider's resource configuration, but they do have difficulty with the automation of the Terraform client through the use of command lines in their workflow.

To simplify the automation of the Terraform workflow, whether for use on a local workstation or in a CI/CD pipeline, we can use **Terragrunt** as a Terraform wrapper that integrates the Terraform workflow.

What we will learn in this recipe is how to use **Terragrunt** (which we already studied in the previous recipe) as a Terraform wrapper.

Getting ready

For this recipe, we must have previously installed the Terragrunt binary on our workstations by following the instructions at `https://terragrunt.gruntwork.io/docs/getting-started/install/#install-terragrunt`.

Before installing Terragrunt, check the version compatibility with Terraform by using the following link: `https://terragrunt.gruntwork.io/docs/getting-started/supported-terraform-versions/`

The Terraform configuration used in this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/demogrunt-wrapper`. It allows us to build resources in Azure. It uses an `env-vars.tfvars` variable file and a remote backend configuration file (`azurerm`) in the `backend.tfvars` file. To create this infrastructure, the following Terraform commands must be executed:

```
terraform init -backend-config="backend.tfvars"
terraform plan  -var-file env-vars.tfvars
terraform apply  -var-file env-vars.tfvars
```

The Terraform configuration for this resource creates resources in Azure, but what we will study in this recipe applies to any Terraform configuration.

The purpose of this recipe is to use the **Terragrunt** configuration to help execute these Terraform commands in an automation context.

How to do it...

Perform the following steps to use **Terragrunt** as a Terraform CLI wrapper:

1. Inside the folder that contains the Terraform configuration, create a new file called `terragrunt.hcl`.
2. In this file, add the following configuration section to configure the `init` command:

```
extra_arguments "custom_backend" {
  commands = [
    "init"
  ]

  arguments = [
    "-backend-config", "backend.tfvars"
  ]
}
```

3. Add the following code to configure the `plan` and `apply` commands:

```
extra_arguments "custom_vars-file" {
  commands = [
    "apply",
    "plan",
    "destroy",
    "refresh"
  ]
   arguments = [
      "-var-file", "env-vars.tfvars"
    ]

}
```

4. In the command-line terminal, from the folder that contains the Terraform configuration, run the following Terragrunt command to initialize the Terraform context:

```
terragrunt init
```

5. Finally, run the following Terragrunt commands to apply the changes we've made:

   ```
   terragrunt plan
   terragrunt apply
   ```

How it works...

In *Step 1*, we created the `terragrunt.hcl` file, which will contain the **Terragrunt** configuration of the Terraform wrapper. In *Step 2*, we described in this file the Terraform execution configuration for the `init` command. In the list of commands, we indicate that this configuration applies to the `init` command, and in the list of arguments, we put an entry for the `--backend-config` option, which takes as a value the `backend.tfvars` file.

Then, in *Step 3*, we did the same configuration operation for the `plan` and `apply` commands. In this configuration, we specify the list of commands: `plan`, `apply`, `destroy`, and `refresh`. We also specified the `-var-file`, `env-vars.tfvars` Terraform CLI options on the arguments block.

Once this configuration file is finished being written, we use it to run **Terragrunt**. In *Step 4*, we execute the `terragrunt init` command, which will use the configuration we wrote and so **Terragrunt** will execute the following command:

```
terraform init -backend-config="backend.tfvars"
```

Finally, to preview the changes, we execute the `terragrunt plan` command, which will cause **Terragrunt** to use the configuration we wrote, and **Terragrunt** will execute the following command:

```
terraform plan -var-file env-vars.tfvars
```

If these changes correspond to your expectations, you can use the following **Terragrunt** command to apply these changes:

```
terragrunt apply
```

Finally, to destroy all the resources provisioned by **Terragrunt**, run the command `terragrunt destroy`, which runs the Terraform command `terraform destroy -var-file env-vars.tfvars`.

See also

- Detailed CLI configuration documentation is available at `https://terragrunt.gruntwork.io/docs/features/keep-your-cli-flags-dry/`.

Generating a self-signed SSL certificate using Terraform

In this recipe, we will learn how to generate an SSL certificate using Terraform.

Let's get started!

Getting ready

This recipe doesn't require any software installation.

The goal of this recipe is to learn to generate a self-signed SSL certificate with Terraform configuration.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/cert`.

How to do it…

To generate an SSL certificate with Terraform, perform the following steps:

1. In a new `main.tf` file, write the following Terraform configuration:

```
terraform {
  required_providers {
    tls = {
      source  = "hashicorp/tls"
      version = "4.0.4"
    }
  }
}

resource "tls_private_key" "private_key" {
  algorithm = "RSA"
}

resource "tls_self_signed_cert" "self_signed_cert" {
  private_key_pem = tls_private_key.private_key.private_key_pem

  validity_period_hours = 120
```

```
    subject {
      common_name = "test.com"
    }

    allowed_uses = [
      "digital_signature",
      "cert_signing",
      "crl_signing",
    ]
  }
```

2. Then, run the basic Terraform workflow by running the commands `terraform init`, plan, and `apply`.

How it works...

In *Step 1* of this recipe, we write Terraform configuration that generates a self-signed SSL certificate. In this configuration, we write the following elements:

- In the provider configuration, we use the provider `hashicorp/tls`.
- Then, we use the resource `tls_private_key` to generate the certificate private key by using the algorithm `RSA`; other algorithm values are listed here: `https://registry.terraform.io/providers/hashicorp/tls/latest/docs/resources/private_key#algorithm`.
- Finally, we use the resource `tls_self_signed_cert`. By using a private key in the preceding resource, `tls_private_key`, we indicate the validity of the certificate is 120 hours, and the common name of the certificate is `test.com` for the sample. The documentation for this resource is available here: `https://registry.terraform.io/providers/hashicorp/tls/latest/docs/resources/self_signed_cert`.

After executing the `terraform apply` command in *Step 2*, all certificate details are stored in the Terraform state.

There's more...

In this recipe, we learned about the basic Terraform configuration to generate a self-signed certificate. The following are some use cases that we can perform to go deeper with the two following use-cases:

1. In this recipe, the content that we generated is inside the terraform state. If we want to generate a key and pem file on the disk, add the following Terraform configuration in main.tf (created in the recipe):

```
resource "local_file" "ca_key" {
  content  = tls_private_key.private_key.private_key_pem
  filename = "${path.module}/ca.key"
}

resource "local_file" "ca_cert" {
  content  = tls_self_signed_cert.self_signed_cert.cert_pem
  filename = "${path.module}/ca.pem"
}
```

   With this above configuration, after the execution of the terraform apply command, the files ca.key and ca.pem will be generated inside the folder that contains this Terraform configuration.

2. In this main recipe, we learned about certificate generation in **PEM** format. We can also use Terraform to generate Windows certification in **PXF** format, based on the generated PEM certificate. To generate a **PXF** certificate, add the following Terraform configuration to main.tf:

   Complete the terraform block with the reference to the pkcs_12 provider:

```
terraform {
  required_providers {
    …
    pkcs12 = {
      source = "chilicat/pkcs12"
      version = "0.0.7"
    }
  }
}
```

   Then, add the Terraform resources:

```
resource "random_password" "self_signed_cert" {
  length  = 24
  special = true
}

resource "pkcs12_from_pem" "self_signed_cert_pkcs12" {
  cert_pem        = tls_self_signed_cert.self_signed_cert.cert_pem
  private_key_pem = tls_private_key.private_key.private_key_pem
  password        = random_password.self_signed_cert.result
}

resource "local_file" "result" {
  filename = "${path.module}/ca.pxf"
  content_base64     = pkcs12_from_pem.self_signed_cert_pkcs12.
result
}
```

In this above configuration, we use the resource `pkcs12_from_pem` to generate the **PFX** based on the **PEM** certificate using a random password and use the resource `local_file` to save the content of the **PFX** certificate to the disk.

With this above configuration, after the execution of the `terraform apply` command, the file `ca.pxf` will be generated inside the folder that contains this Terraform configuration.

See also

- The `tls` Terraform provider documentation is available here: `https://registry.terraform.io/providers/hashicorp/tls/latest/docs`.
- The pkcs12 Terraform provider documentation is available here: `https://registry.terraform.io/providers/chilicat/pkcs12/latest`.
- Article for PXF certificate on Azure: `https://blog.xmi.fr/posts/tls-terraform-azure-self-signed/`.
- GCP documentation on using a certificate with Terraform: `https://cloud.google.com/certificate-authority-service/docs/using-terraform`.
- Renew a certificate using Terraform on AWS: `https://www.missioncloud.com/blog/how-to-generate-and-renew-an-ssl-certificate-using-terraform-on-aws`.
- Blog post on a SSL certificate using Terraform `https://amod-kadam.medium.com/create-private-ca-and-certificates-using-terraform-4b0be8d1e86d`.

Checking the configuration before committing code using Git hooks

Git hooks are scripts that run automatically before or after certain Git events, such as committing code. These scripts can be used to automate tasks and ensure code quality.

In different chapters of this book, we have learned some commands and tools in Terraform that allow us to do Terraform code analysis.

Here are some of the Terraform commands and tools we have learned about:

- `terraform fmt` to format the configuration with the right code indentation
- `terraform validate` to validate the Terraform configuration syntax
- `Tflint`, a linter for Terraform configuration
- `Tfsec` to check some security compliance

The goal of this recipe is to show how to use Git hooks to integrate these commands and the execution of these tools before committing the configuration to a Git repository.

Let's get started!

Getting ready

Before performing this recipe, it is recommended to read the documentation on Git Hooks, available here – `https://www.atlassian.com/git/tutorials/git-hooks` – and also here – `https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`.

You'll need to install the following requirements:

- `pre-commit`, which is the tool for managing pre-commit Git hooks. The installation documentation is available here: `https://pre-commit.com/#install`.
- `Tflint` – the installation documentation is available here: `https://github.com/terraform-linters/tflint`.
- `Tfsec` – the documentation installation is available here: `https://aquasecurity.github.io/tfsec/v1.28.1/guides/installation/`. Also, read the dedicated recipe *Using Tfsec to analyze the compliance of the Terraform configuration* in *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*.

The goal of this recipe is to integrate pre-commit Git Hooks into the GitHub repository that contains the complete source code of this book (`https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition`). To check the configuration formatting, run the `terraform validate` command, execute `tflint`, and run `tfsec` automatically before the commit.

How to do it...

To integrate pre-commit hooks into the Terraform configuration, perform the following steps:

1. In the root of the repository that contains the Terraform configuration, create a new file named `.pre-commit-config.yaml`.
2. In this file, write the following content:

```
repos:
  - repo: https://github.com/antonbabenko/pre-commit-terraform
    rev: "v1.78.0"
    hooks:
      - id: terraform_fmt
      - id: terraform_tflint
      - id: terraform_validate
      - id: terraform_tfsec
```

   The preceding code is simply basic code. The real code in the root of the GitHub repository contains more options to excludes some folders.

3. Then, to manually run the pre-commit based on the YAML configuration above, in the terminal console, run the following pre-commit command:

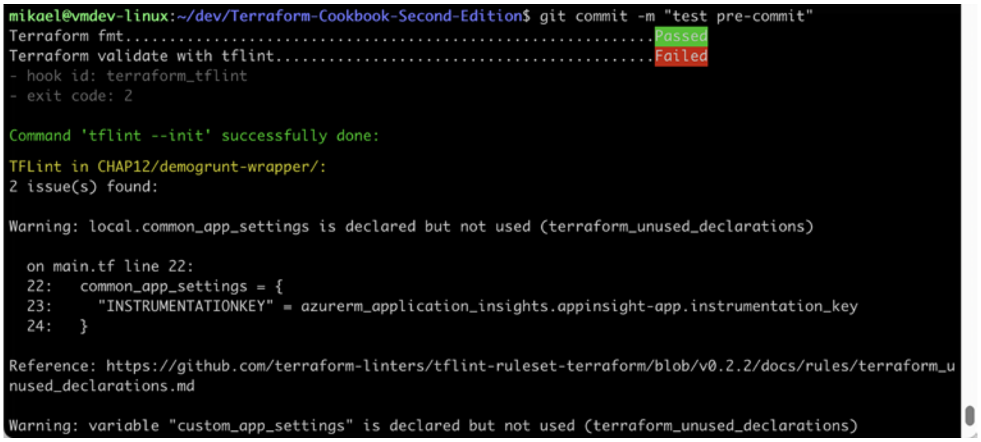
```
pre-commit run --all-files
```

4. Finally, to try the pre-commit Git hooks, create a new Terraform configuration inside the new file `main.tf` in the `precommit-demo` folder.
5. In this file, write the Terraform configuration. The content of this file is not very relevant to this recipe. You can find the content of this file here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/precommit-demo`.
6. Commit this code in Git by running the following commands:

```
git add .
git commit -m "test pre-commit"
```

Note that here, for the first execution of the pre-commit, we must also commit the pre-commit configuration. For committing after the first execution, however, to run the previous commands you need to have files for commits.

The following image shows the partial execution output of this command:



```
mikael@vmdev-linux:~/dev/Terraform-Cookbook-Second-Edition$ git commit -m "test pre-commit"
Terraform fmt...........................................................Passed
Terraform validate with tflint..........................................Failed
- hook id: terraform_tflint
- exit code: 2

Command 'tflint --init' successfully done:

TFLint in CHAP12/demogrunt-wrapper/:
2 issue(s) found:

Warning: local.common_app_settings is declared but not used (terraform_unused_declarations)

  on main.tf line 22:
  22:   common_app_settings = {
  23:     "INSTRUMENTATIONKEY" = azurerm_application_insights.appinsight-app.instrumentation_key
  24:   }

Reference: https://github.com/terraform-linters/tflint-ruleset-terraform/blob/v0.2.2/docs/rules/terraform_u
nused_declarations.md

Warning: variable "custom_app_settings" is declared but not used (terraform_unused_declarations)
```

Figure 12.10: Pre-commit hooks with the Terraform check

We can see that the `git commit` command triggers the pre-commit Git hook execution, starting with `terraform fmt` and `terraform validate`.

How it works...

In *Step 1* and *Step 2*, we create a new `.pre-commit-config.yaml` file at the root of the repository. In this file, we write the YAML configuration that describes the list of tools and commands to run just before the Git commit operation.

Here, in our YAML configuration, we specify the GitHub repo of scripts to run. This GitHub repository – `https://github.com/antonbabenko/pre-commit-terraform` – contains a lot of Terraform commands and tools to integrate into the pre-commit.

In this recipe, we list 4 commands by ID (these IDs are specified in the above GitHub repository): `terraform_fmt`, `terraform_tflint`, `terraform_validate`, and `terraform_tfsec`.

Read the documentation inside the GitHub repository – `https://github.com/antonbabenko/pre-commit-terraform` – to see all included commands and tools. If we want to integrate other tools, add them to the YAML configuration by following the pre-commit documentation available here: `https://pre-commit.com/`.

In *Step 3*, to check that the configuration is working fine, and get the pre-commit result execution, we run the command `pre-commit run is --all-file` to execute the pre-commit on all files present in the repository.

In *Steps 4* and *5*, we test the execution of the pre-commit hook by adding a new Terraform configuration and performing a `git commit` operation. In the resulting command output, we can see that the pre-commit has been triggered.

If the execution of the pre-commit hooks returns an error, the files are not committed, which allows us to commit only files that respect the rules of validation and compliance specified in the commands and tools integrated into the pre-commit.

There's more...

If for some reason we want to disable the pre-commit hooks, we can add the `–no-verify` option to the `git commit` command, as explained in this blog post: `https://ma.ttias.be/git-commit-without-pre-commit-hook/`.

We have learned in this recipe how, using pre-commit Git hooks, we can execute check and validation commands before committing code, but of course this does not prevent us from also integrating these commands and tools into the Terraform CI/CD pipelines, which we will see in detail in *Chapter 13, Automating Terraform Execution in a CI/CD Pipeline*.

See also

- The `pre-commit` documentation is available here: `https://pre-commit.com/`.
- The GitHub repository for the Terraform pre-commit is available at `https://github.com/antonbabenko/pre-commit-terraform`.
- A blog post on the Terraform pre-commit is available here: `https://jamescook.dev/pre-commit-for-terraform`.

Visualizing Terraform resource dependencies with Rover

In *Chapter 6*, *Applying a Basic Terraform Workflow* in the *Generating the dependency graph recipe*, we learned how to generate a Terraform dependency graph using the Terraform command line.

This generated graph provides a global visualization of the dependencies of modules and resources, but it is static and sometimes difficult to read due to the complexity of the graph.

Among the open-source tools that integrate with Terraform, there is a tool called Rover, which is an open-source tool that allows for dynamic and interactive visualization of the dependency graph of the Terraform resources that are provisioned.

In this recipe, we will learn how to use Rover to visualize the dependency graph.

Let's get started!

Getting ready

We will just install Rover before performing the recipe.

To install Rover, refer to the documentation here: `https://github.com/im2nguyen/rover`. We can use it by installing the CLI locally or by using Docker.

We do not use a specific Terraform configuration for this recipe. For the sample configuration, we will use the existing Terraform configuration, which is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP08/sample-app`.

How to do it…

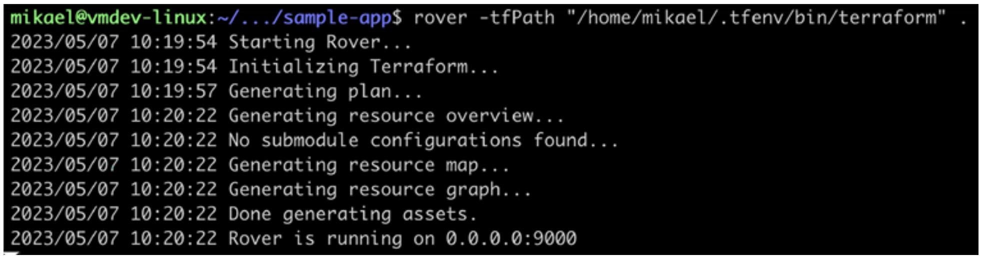
To visualize the dependency graph with Rover, perform the following steps:

1. Navigate into the folder that contains the Terraform configuration to display the graph and run the command `terraform init`.
2. Run the command `terraform plan -out="tfplan.out"`.
3. Then, run Rover on this generated plan by running this command:

```
rover -tfPath "/home/mikael/.tfenv/bin/terraform"
```

In the above command, the tfPath parameter value was set for Linux. For Windows users, you can set -tfPath "`terraform.exe`", or the specific path for `terraform.exe`.

The following image shows the output of the execution of this command:



```
mikael@vmdev-linux:~/.../sample-app$ rover -tfPath "/home/mikael/.tfenv/bin/terraform" .
2023/05/07 10:19:54 Starting Rover...
2023/05/07 10:19:54 Initializing Terraform...
2023/05/07 10:19:57 Generating plan...
2023/05/07 10:20:22 Generating resource overview...
2023/05/07 10:20:22 No submodule configurations found...
2023/05/07 10:20:22 Generating resource map...
2023/05/07 10:20:22 Generating resource graph...
2023/05/07 10:20:22 Done generating assets.
2023/05/07 10:20:22 Rover is running on 0.0.0.0:9000
```

Figure 12.11: Rover execution

Rover will start a website on port `9000` on `localhost`.

4. Open your browser and navigate to the URL `http://localhost:9000`. The Rover website displays an interactive dependency graph, with variables, resources, outputs, and more details.

   The following image shows the content of the generated Rover page:

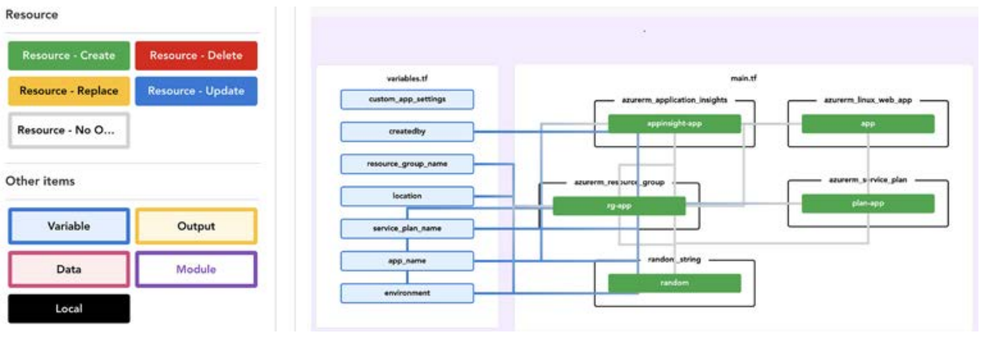


Figure 12.12: Rover dependency graph

We can see the dependency graph, with variables, and in the left-hand menu, some options to filter the data in the graph.

How it works...

In *Steps 1* and *2* of this recipe, we run the command `terraform init` and generate the output file of the `terraform plan` of our Terraform configuration.

Then, we use the Rover command line `rover -tfPath "/home/mikael/.tfenv/bin/terraform".` to generate a dynamic and more user-friendly UI of the dependency graph.

On this command line, we add the optional argument `-tfPath` to specify the path of the Terraform binary. And we specify the path of the generated plan output with the "." to indicate that is in the current folder.

For more information about all arguments available on the Rover command line, execute the command `rover --help`.

Finally, we open the `localhost:9000` page and we can see the dependency graph generated by Rover.

See also

- The documentation for Rover is available here `https://github.com/im2nguyen/rover`.
- See Rover in action in these videos – `https://www.youtube.com/watch?v=2Y9yXgURxyE` – and here – `https://www.hashicorp.com/resources/terraform-plan-interactive-configuration-and-state-visualization-with-rover`.

Using the Terraform CDK for developers

So far in this book, we have seen how to write Terraform configuration using HCL. HCL is a configuration language that has the advantage of being easy to read for humans. However, application developers who wanted to use Terraform had to learn a new language.

To address this issue, in August 2022, HashiCorp announced the availability of a **Cloud Development Kit for Terraform** (**CDKTF**). Read the announcement blog post here: `https://www.hashicorp.com/blog/cdk-for-terraform-now-generally-available`.

CDKTF is a tool that allows you to define IaC using familiar programming languages such as TypeScript, JavaScript, Python, and Java.

With CDKTF, you write code that describes your infrastructure in a higher-level language, allowing for more abstraction layers compared to the Terraform (HCL) language. CDKTF synthesizes your code, i.e., generates the equivalent Terraform configuration under the hood. This allows you to use a familiar programming language syntax to define your infrastructure, while still leveraging the power and flexibility of the Terraform engine to provision and manage resources.

Some of the benefits of using CDKTF include:

- **Simplicity**: CDKTF makes it easy to create and manage infrastructure with just a few lines of code, using a high-level object-oriented syntax.
- **Reusability**: CDKTF makes it easy to create reusable components, which can be shared across different projects or teams.

- **Type Safety**: CDKTF allows you to leverage the type checking and error reporting features of modern programming languages to catch errors at compile time, reducing the risk of runtime errors and speeding up the development cycle.
- **Ease of Integration**: Because CDKTF generates standard Terraform code, it can be used seamlessly with other Terraform tools and workflows.
- **Faster Development**: Assuming you choose a language you're already familiar with, CDKTF could allow you to create and modify infrastructure quicker than with traditional Terraform code, resulting in faster development cycles and more responsive infrastructure.

In this recipe, we will learn how to use CDKTF using the TypeScript language and will not use any other languages such as Go or C#.

Let's get started!

Getting ready

To complete this recipe, you'll need to install:

- NodeJS and npm – the installation documentation is available here: `https://nodejs.org/en/download`.
- TypeScript: Download and install using the documentation here: `https://www.typescriptlang.org/download`.

You need to also have some knowledge of TypeScript/Node.js development practices.

For the sample of this recipe, using the CDKTF tool, we will provision an Azure infrastructure composed of an Azure Resource Group, an Azure Service Plan, and an Azure Linux App Service.

The recipe will be composed of two parts:

- Writing the TypeScript code of the infrastructure
- Using the CDKTF CLI to provision the infrastructure

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/cdktf-demo`.

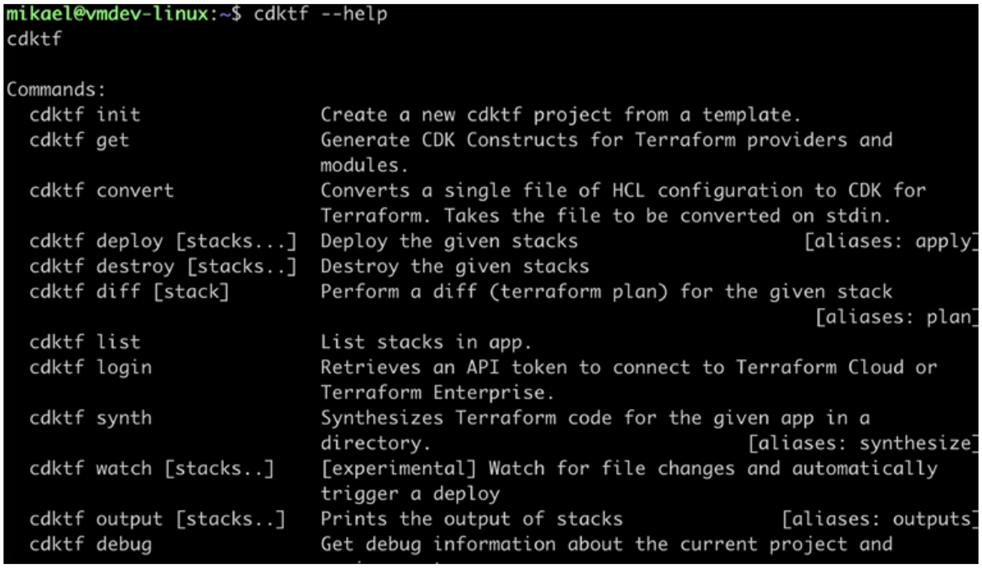
How to do it...

To use CDKTF, perform the following steps:

1. Install the npm package `cdktf-cli` available here – `https://www.npmjs.com/package/cdktf-cli` – by executing the following command:

   ```
   npm install --global cdktf-cli@latest
   ```

   Then, check the package installation by running the command `cdktf-help`:

   ```
   mikael@vmdev-linux:~$ cdktf --help
   cdktf

   Commands:
     cdktf init                 Create a new cdktf project from a template.
     cdktf get                  Generate CDK Constructs for Terraform providers and
                                modules.
     cdktf convert              Converts a single file of HCL configuration to CDK for
                                Terraform. Takes the file to be converted on stdin.
     cdktf deploy [stacks...]   Deploy the given stacks                    [aliases: apply
     cdktf destroy [stacks..]   Destroy the given stacks
     cdktf diff [stack]         Perform a diff (terraform plan) for the given stack
                                                                            [aliases: plan
     cdktf list                 List stacks in app.
     cdktf login                Retrieves an API token to connect to Terraform Cloud or
                                Terraform Enterprise.
     cdktf synth                Synthesizes Terraform code for the given app in a
                                directory.                            [aliases: synthesize
     cdktf watch [stacks..]     [experimental] Watch for file changes and automatically
                                trigger a deploy
     cdktf output [stacks..]    Prints the output of stacks              [aliases: outputs
     cdktf debug                Get debug information about the current project and
   ```

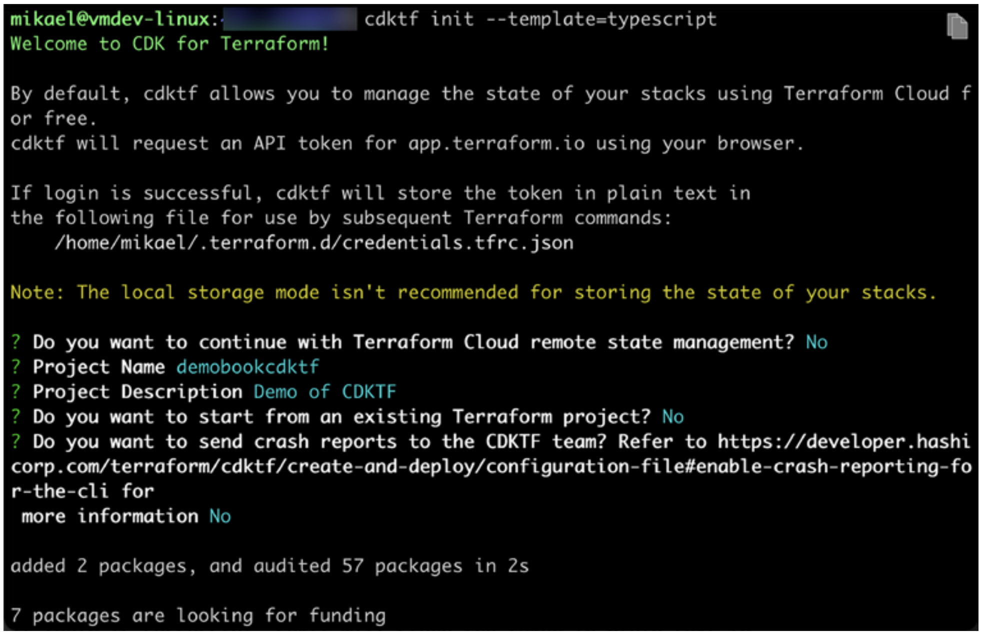
   Figure 12.13: cdktf list commands

   We can see that this above command displays all CDKTF CLI available commands.

2. Create a new folder that will contain the code, named `cdktf-demo` (this name is an example for this recipe).
3. Inside this folder, `cdktf-demo`, in the terminal console, run this command:

   ```
   cdktf init --template=typescript
   ```

Then, during this execution, the CDKTF CLI will ask some questions to provide the best starting template, as shown in the following image:



```
mikael@vmdev-linux:[redacted] cdktf init --template=typescript
Welcome to CDK for Terraform!

By default, cdktf allows you to manage the state of your stacks using Terraform Cloud f
or free.
cdktf will request an API token for app.terraform.io using your browser.

If login is successful, cdktf will store the token in plain text in
the following file for use by subsequent Terraform commands:
    /home/mikael/.terraform.d/credentials.tfrc.json

Note: The local storage mode isn't recommended for storing the state of your stacks.

? Do you want to continue with Terraform Cloud remote state management? No
? Project Name demobookcdktf
? Project Description Demo of CDKTF
? Do you want to start from an existing Terraform project? No
? Do you want to send crash reports to the CDKTF team? Refer to https://developer.hashi
corp.com/terraform/cdktf/create-and-deploy/configuration-file#enable-crash-reporting-fo
r-the-cli for
 more information No

added 2 packages, and audited 57 packages in 2s

7 packages are looking for funding
```

Figure 12.14: cdktf init command

We can see folders and files of the templates in the `cdktf-demo` folder.

4. In the terminal console, run the command `npm install @cdktf/provider-azurerm`.
5. Then, inside the folder `cdktf-demo`, in the generated code, delete the code of the `main.ts` and replace it with the following code (which we explain in detail in the *How it works…* section):

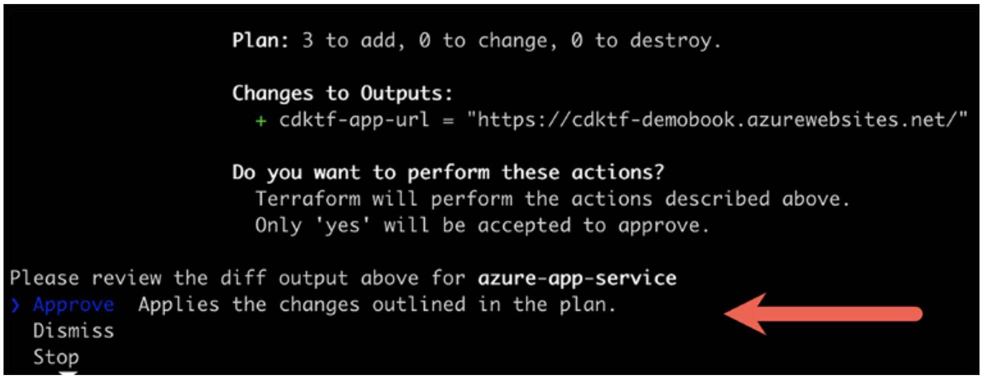
```
import { Construct } from "constructs";
import { App, TerraformStack, TerraformOutput } from "cdktf";
import { AzurermProvider } from "@cdktf/provider-azurerm/lib/
provider";
import { ResourceGroup } from "@cdktf/provider-azurerm/lib/resource-
group";
import { ServicePlan } from "@cdktf/provider-azurerm/lib/service-
plan";
import { LinuxWebApp } from "@cdktf/provider-azurerm/lib/linux-web-
app";
```

```
class CDKTFDemo extends TerraformStack {
  constructor(scope: Construct, name: string) {
    super(scope, name);

    let random = (Math.random() + 1).toString(36).substring(7);
    new AzurermProvider(this, "azureFeature", {
      features: {},
    });

    const rg = new ResourceGroup(this, "cdktf-rg", {
      name: "cdktf-demobook-",
      location: "westeurope",
    });

    const asp = new ServicePlan(this, "cdktf-asp", {
      osType: "Linux",
      skuName: "S1",
      resourceGroupName: rg.name,
      location: rg.location,
      name: "cdktf-demobook"+random
    });

    const app = new LinuxWebApp(this, "cdktf-app", {
      name: "cdktf-demobook",
      location: rg.location,
      servicePlanId: asp.id,
      resourceGroupName: rg.name,
      clientAffinityEnabled: false,
      httpsOnly: true,
      siteConfig: {
      },
    });

    new TerraformOutput(this, "cdktf-app-url", {
      value: 'https://${app.name}.azurewebsites.net/',
    });
  }
}

const app = new App();
new CDKTFDemo(app, "azure-app-service");
app.synth();
```

6. Finally, to deploy this infrastructure, in the terminal console, run the following `cdktf` command:

```
cdktf deploy
```

During the execution of this command, the CDKTF CLI will ask for confirmation to apply.



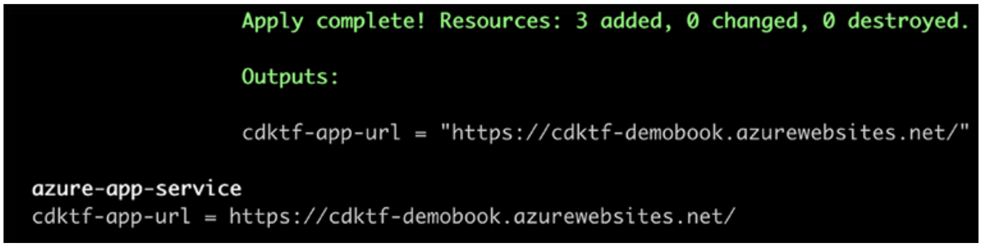
```
                   Plan: 3 to add, 0 to change, 0 to destroy.

                   Changes to Outputs:
                     + cdktf-app-url = "https://cdktf-demobook.azurewebsites.net/"

                   Do you want to perform these actions?
                     Terraform will perform the actions described above.
                     Only 'yes' will be accepted to approve.

Please review the diff output above for azure-app-service
❯ Approve  Applies the changes outlined in the plan.
  Dismiss
  Stop
```

Figure 12.15: cdktf approval of application

Select the **Approve** choice to confirm the application.

The provisioning of the infrastructure will start….

7. At the end of the provisioning, the following command output displays the result of the application execution.



```
                   Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

                   Outputs:

                   cdktf-app-url = "https://cdktf-demobook.azurewebsites.net/"

  azure-app-service
  cdktf-app-url = https://cdktf-demobook.azurewebsites.net/
```

Figure 12.16: cdktf apply execution output

All resources are provisioned, and the terminal displays the value of the output `cdktf-app-service`.

How it works...

In *Step 1*, we install the CDKTF npm package named `cdktf-cli` from the npm registry.

In *Step 2*, we create a new folder that will contain the code of the infrastructure in the TypeScript language.

In *Step 3*, we run the command `cdktf init -template=Typescript` to generate a template for the CDKTF code in the TypeScript language.

Then, in *Step 4*, we install the `azurerm` provider `npm` package for CDKTF by running the command `npm install @cdktf/provider-azurerm`.

The package will be downloaded in the `node_modules` folder, which we can see in the following image:

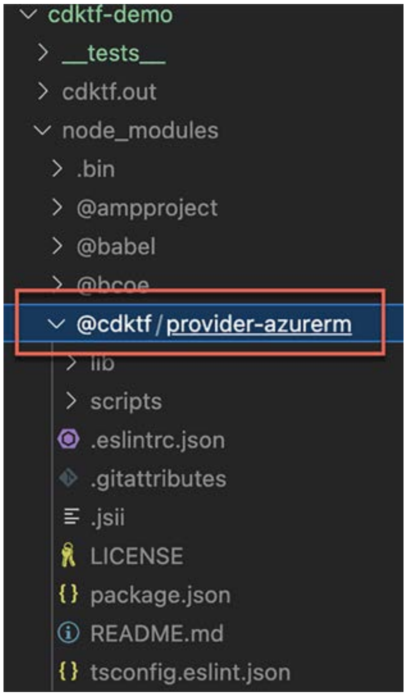


Figure 12.17: cdktf provider-azurerm npm package

We can see the `provider-azurerm` npm package downloaded in the `node_modules` folder.

In *Step 5*, we write the code for our infrastructure in `main.tf` in the TypeScript language. Here are some details of this code:

- We start the code by importing the `azurerm` npm provider library.
- Then, we write the TypeScript code to provision each Azure resource:

  This code below provisions the Azure Resource Group:

```
const rg = new ResourceGroup(this, "cdktf-rg", {
      name: "cdktf-demobook",
      location: "westeurope",
    });
```

  This code below provisions the Azure Service Plan:

```
    const asp = new ServicePlan(this, "cdktf-asp", {
      osType: "Linux",
      skuName: "S1",
      resourceGroupName: rg.name,
      location: rg.location,
      name: "cdktf-demobook"
    });
```

  This code below provisions the Linux App Service:

```
    const app = new LinuxWebApp(this, "cdktf-app", {
      name: "cdktf-demobook",
      location: rg.location,
      servicePlanId: asp.id,
      resourceGroupName: rg.name,
      clientAffinityEnabled: false,
      httpsOnly: true,
      siteConfig: {
      },
    });
```

  Finally, we write the following code to output the URL of the Azure Web App:

```
    new TerraformOutput(this, "cdktf-app-url", {
      value: 'https://${app.name}.azurewebsites.net/',
    });
```

- The last line of the code is the main code that calls the class and CDKTF function with the following code:

  ```
  const app = new App();
  new CDKTFDemo(app, "azure-app-service");
  app.synth();
  ```

In *Step 6*, we execute the command `cdktf deploy`, which performs the following operations:

- Generate the HCL code from the above TypeScript code. This HCL code (in JSON format) is stored in the `cdktf.out` subfolder, and this operation is called synthesizing.
- Inside this generated HCL code, run the `terraform init` command.
- Inside this generated HCL code, run the `terraform apply` command.
- Then, exactly as would happen with the `terraform apply` command, the execution of this `cdktf` command asks for confirmation to apply the changes of code.
- We select the option `Approve` to apply the changes and provision the resources.

In *Step 7*, we see the output of the `cdktf deploy` command.

There's more...

Here in this recipe, we learned the basic steps for using CDKTF, and we explicitly used only one CDKTF command to apply the resources.

There are other CDKTF CLI commands that can be useful. Below are some of them:

- Run the command `cdktf diff` to get a preview of changes. This command is equal to the `terraform plan` command.
- Run the command `cdktf destroy` to delete all provisioned resources. This command is equal to the `terraform destroy` command.
- In `cdktf deploy`, add the option `-auto-approve` to directly apply the changes and not be asked for confirmation. This option can be used in automated mode.

Additionally, if we have already written Terraform configuration and we want to convert it into a CDKTF language (for example, into TypeScript), we can run the `cdktf convert` command, for example:

```
cat main.tf | cdktf convert --provider hashicorp/azurerm > importedmain.ts
```

Also, if we want to use only the synthesize operation to generate the CDKTF TypeScript code in the HCL language, run the command `cdktf synth`; the HCL configuration will be generated in the `cdktf.out` folder.

Finally, we can write and execute integration testing using the developer language that we use in CDKTF; in our case, TypeScript using the Jest test framework. For more information about CDKTF testing, read the documentation here: `https://developer.hashicorp.com/terraform/cdktf/test/unit-tests`.

See also

- Documentation on CDKTF is available here: `https://developer.hashicorp.com/terraform/cdktf`.
- CDKTF's installation documentation is available here: `https://developer.hashicorp.com/terraform/tutorials/cdktf/cdktf-install`.
- A sample of CDKTF for Azure is available here: `https://github.com/hashicorp/terraform-cdk/blob/main/examples/typescript/azure-app-service/main.ts`.
- A video demonstration of CDKTF: `https://www.youtube.com/watch?v=bssG1piyaKw`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

13

Automating Terraform Execution in a CI/CD Pipeline

In the previous chapters of this book, we executed all Terraform commands locally in a console with occasional manual user interactions to check the plan provided by Terraform.

In this chapter, we will learn how to run Terraform automatically.

What does it mean to run Terraform automatically?

As we know, in the basic Terraform workflow, there are the `apply` and `destroy` commands, which ask the user to confirm the application of changes.

However, in a DevOps and automation context, our Terraform configuration should be executed in a CI/CD pipeline that does not require user interaction.

Note: Even if we talk about automation, the step of applying `terraform plan` should still require user confirmation. The automation concerns user confirmation during the execution of `terraform apply` (or `destroy`).

As a reminder, we have already learned one of the ways to automate the execution in *Chapter 10*, *Using Terraform for Docker and Kubernetes Deployment*, in the recipe *Applying GitOps practices on Kubernetes with Terraform*, in which our Terraform configuration is executed in a GitOps context.

In this chapter, we will learn how to integrate Terraform execution for Terraform configuration and Terraform modules in CI/CD pipelines in two DevOps cloud services:

- GitHub
- Azure Pipelines

We will also learn in this chapter how to run Terraform in an automation context and how to run commands or scripts to display a summary of the results of `terraform plan`.

Note: What we will learn in these two services can be applied in all other CI/CD systems, such as Jenkins, GitLab, and Bitbucket.

Before reading this chapter, please review the best practices for migrating from manual provisioning processes to a collaborative IaC workflow at `https://developer.hashicorp.com/terraform/cloud-docs/recommended-practices/part3`.

In this chapter, we will cover the following recipes:

- Running Terraform in automation mode
- Displaying a summary of the execution of `terraform plan`
- Building CI/CD pipelines to apply Terraform configurations in Azure Pipelines
- Automating Terraform execution in GitHub Actions
- Working with workspaces in CI/CD
- Building CI/CD for Terraform modules in Azure Pipelines
- Building a workflow for publishing Terraform modules using GitHub Actions

Running Terraform in automation mode

To start this chapter, in this recipe, we will explain how to enable Terraform to run automatically without user interaction using three different solutions.

Let's get started!

Getting ready

There are no technical requirements to complete this recipe.

Before learning how to integrate the Terraform execution into a CI/CD pipeline (as we will see in other recipes in this chapters), it's important to understand how to run Terraform in automation mode locally by manipulating the Terraform CLI options.

The goal of this recipe is to show how to run Terraform automatically using different options in the command line.

How to do it…

To run Terraform in automation mode, we have multiple solutions, which are detailed below:

- The first point of automation concerns the entry of the values of the variables during the plan. If our Terraform configuration requires the input of the values of the variables during the execution of `terraform plan`, we can alleviate the need to manually input the values by adding the argument `-input=false` in the `terraform plan` command, by also adding the argument `-var` or `-var-file`, as below:

  ```
  terraform plan -var-file=dev.tfvars -input=false
  ```

  For more details about the use of `-input` arguments, read the documentation here: `https://developer.hashicorp.com/terraform/cli/commands/plan#input-false`

  And for more details about the `-var` and `-var-file` arguments, read the recipe *Manipulating variables* in *Chapter 2*, *Writing Terraform Configurations*.

- The second way is to use `terraform plan` out using the following Terraform command workflow:

  ```
  terraform init
  terraform plan -out=tfplan.out
  terraform apply tfplan.out
  ```

  In the preceding commands, we generate the `terraform plan` result in the `tfplan.out` file and we use this file in the input of the `terraform apply` command.

  The execution of `terraform apply` will be done without any user confirmation. Applying a saved `plan` file is highly recommended because, in addition to including the automation, it ensures the exact application of what was generated in the `terraform plan`. And thus, if the Terraform configuration has been modified between the `terraform plan` and the `terraform apply` (example committed in a Git repository) and it is this `terraform plan` that has been validated, it is thus ensured that it is the validated plan that will be applied or else an error will be shown. However, in order to apply this solution, it is strongly recommended that the `terraform plan` and the `terraform apply` be executed on the same machine and the `apply` must be able to access the file generated during the `terraform plan`.

- Another way to automate the execution of the Terraform workflow is to add the argument `-auto-approve` to the `terraform apply` command, as shown here:

  ```
  terraform init
  terraform plan
  terraform apply -auto-approve
  ```

  By adding this argument, Terraform will not ask the user for confirmation to perform the `terraform apply` of the changes mentioned in the previous execution of the `terraform plan`. Be careful, using this solution is not recommended because if changes in the configurations take place between the `terraform plan` and the `terraform apply`, then uncontrolled changes can be applied. So be sure to use the `-auto-approve` argument on the same code version as the plan.

How it works...

In this recipe, we learned three ways to automate the Terraform execution using Terraform CLI options:

- The first way is to skip the prompting of variables, during the `plan` and `apply` execution, by adding the option `-input=false`.
- The second way is to generate `plan` results in a file using the `-out` option and use this `plan` file in the `apply` execution.
- The last way is to automate the approval of the `apply` command by using the `-auto-approve` option.

There's more...

In the case that you run the `terraform plan` and `apply` on different machines (for example, in a CI/CD pipeline in a different runner), follow this documentation to learn the process of how to transfer the configuration between the `plan` and the `apply`: `https://developer.hashicorp.com/terraform/tutorials/automation/automate-terraform#plan-and-apply-on-different-machines`.

See also

- Documentation for running Terraform in automation mode is available here: `https://developer.hashicorp.com/terraform/tutorials/automation/automate-terraform`

Displaying a summary of the execution of terraform plan

In the previous recipe, we learned how to run Terraform in automation mode. The major difference between the workflows that are running locally and the workflows that are running in a CI/CD pipeline is the automation of the `terraform plan` and `apply` commands without any user interaction.

However, it can be very interesting (I would even say necessary) to visualize a summary of what will be applied by Terraform.

In this recipe, we will discuss some tips to display the summary of the `terraform plan` result inside a CI/CD pipeline.

Let's get started!

Getting ready

There are no software requirements to complete this recipe.

How to do it...

The first tip to display a short plan summary result is to display if there are some changes or no changes by using the `detailed-exit-code` argument of the `terraform plan` command by running the following pseudo code:

```
terraform plan -out=tfplan –detailled-exite-code > /dev/null
OUT=$?
//Pseudo code to display information
If $OUT==0 THEN "No changes"
If $OUT==1 THEN "Terraform has failed"
If $OUT==2 THEN "There is changes"
```

For more information about the `-detailed-exit-code` argument, read the documentation here: `https://developer.hashicorp.com/terraform/cli/commands/plan#detailed-exitcode`.

The second way to display the plan summary is to use the `tf-summarize` tool, available here: `https://github.com/dineshba/tf-summarize`.

`tf-summarize` is a tool that displays a short and human-readable summary of the `terraform plan` result.

For this recipe, we will run `tf-summarize` on existing Terraform configurations, which are available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/cert`.

Perform the following steps to use `tf-summarize`:

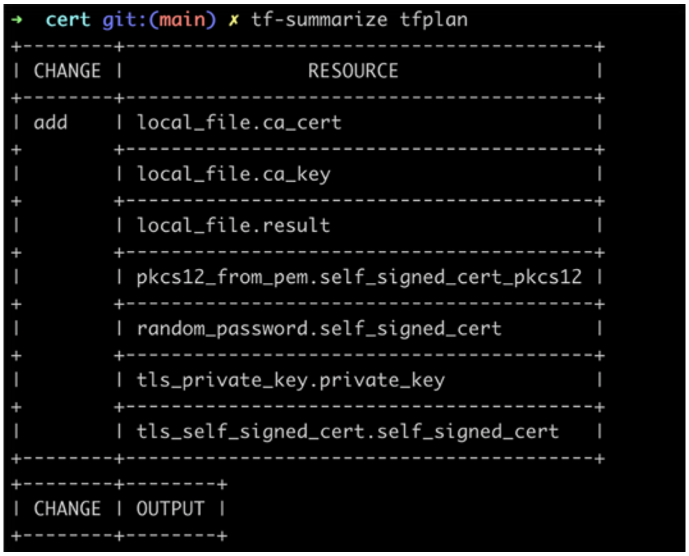
1. Download and install it by referring to the documentation here: `https://github.com/dineshba/tf-summarize#install`.
2. Navigate to the Terraform configuration folder you want to see a plan summary of and run the `terraform init` command and then the following `terraform plan` command:

```
terraform plan -out=tfplan
```

3. Run `tf-summarize` with the following command:

```
tf-summarize tfplan
```

The following image shows the output of the above command:



```
→  cert git:(main) ✗ tf-summarize tfplan
+--------+-----------------------------------------+
| CHANGE |                RESOURCE                 |
+--------+-----------------------------------------+
| add    | local_file.ca_cert                      |
+        +-----------------------------------------+
|        | local_file.ca_key                       |
+        +-----------------------------------------+
|        | local_file.result                       |
+        +-----------------------------------------+
|        | pkcs12_from_pem.self_signed_cert_pkcs12 |
+        +-----------------------------------------+
|        | random_password.self_signed_cert        |
+        +-----------------------------------------+
|        | tls_private_key.private_key             |
+        +-----------------------------------------+
|        | tls_self_signed_cert.self_signed_cert   |
+--------+-----------------------------------------+
+--------+--------+
| CHANGE | OUTPUT |
+--------+--------+
```

Figure 13.1: Terraform plan summary

We can see a table with a list of actions performed on different resources here; in our sample Terraform execution, all resources will be added.

The advantage of `tf-summarize` is that we can display and export the output of the summary result as JSON or Markdown so it is easy to integrate it in your CI/CD system.

To see more examples of the usage of `tf-summarize`, read the documentation here: `https://github.com/dineshba/tf-summarize#examples`.

Another way is to export the result of the `terraform plan` command in JSON format and inspect it using the JQ tool (available here: `https://stedolan.github.io/jq/`). For example, perform the following commands inside the Terraform configuration to display the number of resources that will be created, the number that will be updated, and the number of resources that will be deleted:

```
terraform plan -out=tfplan
terraform show -json tfplan > plan.json
 jq -r '[.resource_changes | select(.[].change.actions[] == "create")] |
length' plan.json
```

In the preceding script, we export the result of the `plan` command in JSON format (line 2) and we use the JQ tool to filter only created resources.

There's more...

In this recipe, we have learned about some different ways of using Terraform commands, to script them, and display some information based on the `terraform plan` summary.

You are free to use your own scripting language and tools to integrate the result of the `terraform plan` according to your CI/CD system.

See also

- A blog post on how to display the `terraform plan` result in Azure Pipelines: `https://chamindac.blogspot.com/2022/08/show-terraform-plan-in-azure-pipeline.html`
- A blog post on how to integrate the result of the `terraform plan` in a pull request using Azure Pipelines: `https://www.natmarchand.fr/terraform-plan-as-pr-comment-in-azure-devops/#more-315`

Building CI/CD pipelines to apply Terraform configurations in Azure Pipelines

In all the recipes in this book, we've discussed Terraform configuration, CLI execution, and their benefits for IaC.

Now, in this recipe, we will discuss how we will integrate this Terraform workflow into a CI/CD pipeline in Azure Pipelines using the Terraform extension for Azure DevOps and Pipelines YAML.

Getting ready

The purpose of this recipe is not to explain in detail how Azure Pipelines works, but just to focus on the execution of Terraform in Azure Pipelines. To learn more about Azure Pipelines, I suggest you look at the official documentation at `https://docs.microsoft.com/en-us/azure/devops/pipelines/index?view=azure-devops`.

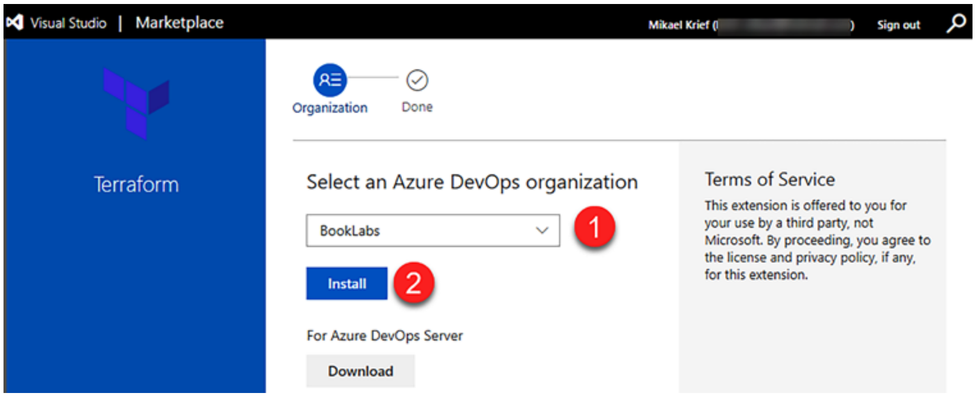
To use Terraform in Azure Pipelines, there are a couple of solutions:

- Use custom scripts (PowerShell and Bash) to execute the Terraform CLI commands.
- Use Terraform extensions for Azure DevOps.

In this recipe, we will learn how to use the Terraform extension for Azure DevOps published by Charles Zipp (with the understanding that there are, of course, other extensions available by other publishers).

To install this extension, implement the following steps:

1. Go to `https://marketplace.visualstudio.com/items?itemName=charleszipp.azure-pipelines-tasks-terraform` in your browser and click on **Terraform Build & Release Tasks**.
2. At the top of the page, click on **Get it free**.
3. On the installation page, in the **Organization** dropdown, choose the organization in which the extension will be installed (1), then click on the **Install** button (2):



This extension is offered to you for your use by a third party, not Microsoft. By proceeding, you agree to the license and privacy policy, if any, for this extension.

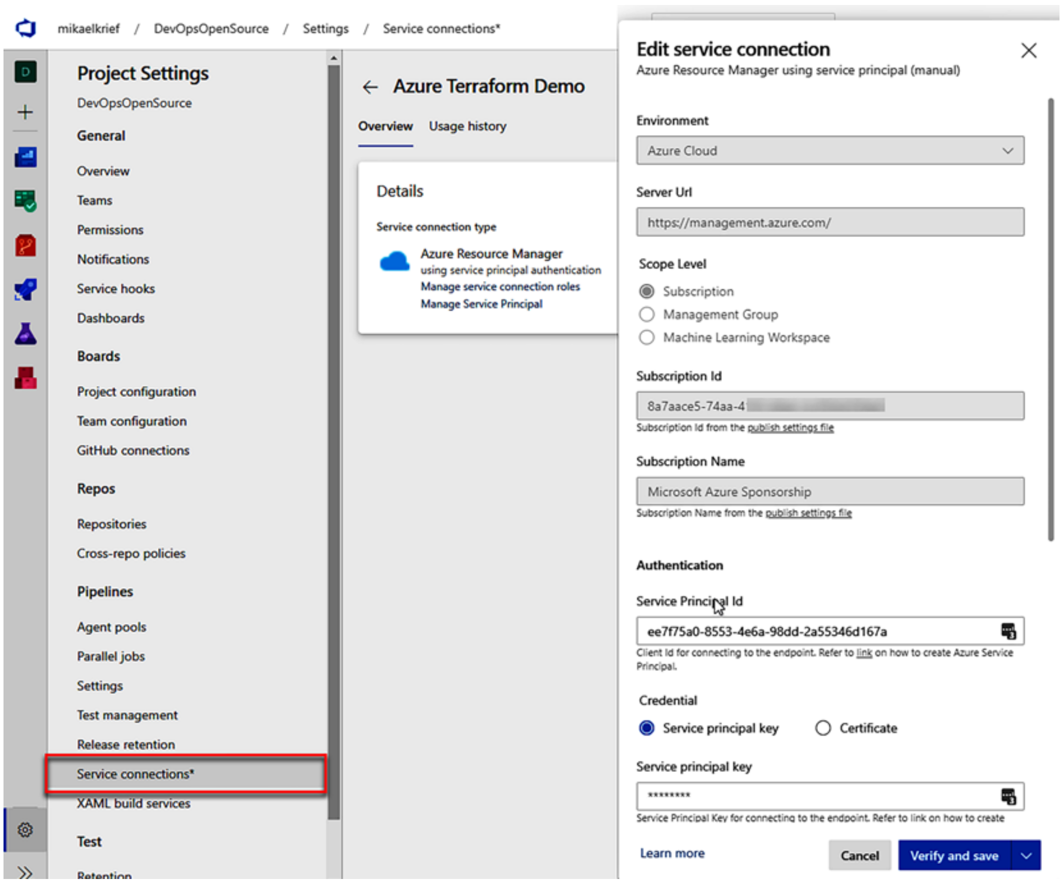
Figure 13.2: Azure DevOps installing the Terraform extension

The extension will be installed in your **Azure DevOps** organization.

Additionally, for the Terraform state, we will use an Azure remote backend. To be able to use it in Azure (Azure Storage, to be precise) with Terraform, we learned in the *Protecting the state file in Azure remote backend* recipe of *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*, that an Azure service principal must be created.

To create this connection with Azure, in Azure Pipelines, we set up a service connection with the information from the created Azure service principal. To operate this, in the **Project Settings**, we navigate to the **Service connections** menu. Then we create a new Azure RM service connection and configure it with the service properties.

The following screenshot shows the service connection to my **Azure Terraform Demo** configuration:



Details

Service connection type

Azure Resource Manager
using service principal authentication
Manage service connection roles
Manage Service Principal

Edit service connection
Azure Resource Manager using service principal (manual)

Environment
Azure Cloud

Server Url
https://management.azure.com/

Scope Level
● Subscription
○ Management Group
○ Machine Learning Workspace

Subscription Id
8a7aace5-74aa-4[redacted]
Subscription Id from the publish settings file

Subscription Name
Microsoft Azure Sponsorship
Subscription Name from the publish settings file

Authentication

Service Principal Id
ee7f75a0-8553-4e6a-98dd-2a55346d167a
Client Id for connecting to the endpoint. Refer to link on how to create Azure Service Principal.

Credential
● Service principal key ○ Certificate

Service principal key
****
Service Principal Key for connecting to the endpoint. Refer to link on how to create

Figure 13.3: Azure DevOps creating Azure service connection

Finally, the code that contains the Terraform configuration must be stored in a Git repository, such as GitHub or Azure Repos (in this recipe, we will use GitHub as the code repository that contains the Terraform configuration).

Note that, in this recipe, we will not study the deployed Terraform configuration code, which is very basic (it generates a self-signed certificate, which we already covered in *Chapter 12*, *Deep-Diving into Terraform*, in the recipe *Generate a self-signed certificate with Terraform*) – its purpose is to demonstrate the implementation of the pipeline.

The Terraform configuration source code that will be used in this recipe is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP13/azpipeline`.

How to do it...

To create the pipeline, we perform the following steps:

1. In the **Azure DevOps** menu, click **Pipelines**:

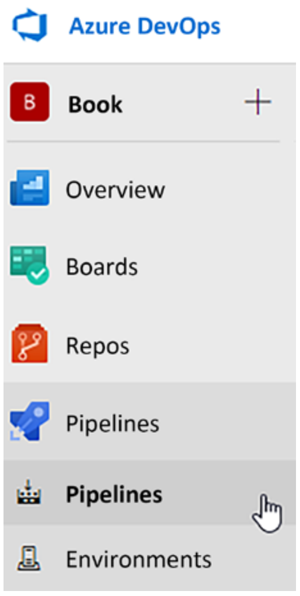
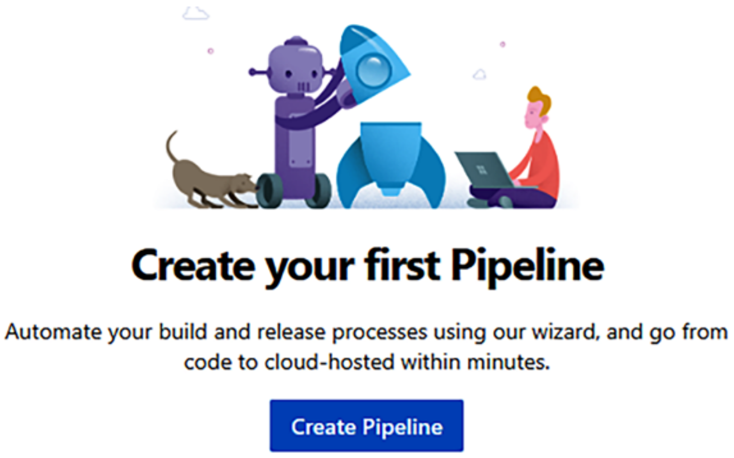


Figure 13.4: Azure DevOps Pipelines menu

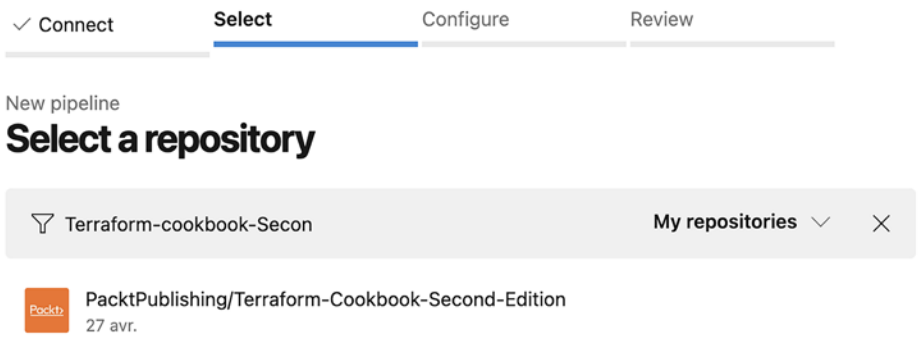
2. Click on the **Create Pipeline** button:


Create your first Pipeline

Automate your build and release processes using our wizard, and go from code to cloud-hosted within minutes.

Figure 13.5: Azure DevOps Create Pipeline

3. For the code source, select the Git repository that contains the Terraform configuration. For this recipe, we choose our GitHub repository and select the **Starter** pipeline option to start with a new pipeline from scratch:



New pipeline

Select a repository

Terraform-cookbook-Secon

PacktPublishing/Terraform-Cookbook-Second-Edition
27 avr.

Figure 13.6: Azure DevOps select repository for pipeline

4. The pipeline editor opens, and you can start writing the CI/CD steps directly online. Let's look at the code for this pipeline, which is in YAML format. First, we're going to configure the pipeline options with the following code to use an Ubuntu agent:

```yaml
trigger:
  - master
pool:
  vmImage: 'ubuntu-latest'
```

5. Then, we tell the pipeline to download the desired version of the Terraform binary by adding this code:

```
- task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-installer.TerraformInstaller@0
  displayName: 'Install Terraform 1.4.4'
  inputs:
    terraformVersion: 1.4.4
```

6. We continue with the first command of the Terraform workflow and execute the `terraform init` command:

```
- task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-cli.TerraformCLI@0
  displayName: 'terraform init'
  inputs:
    command: init
    workingDirectory: "CHAP12/cert/"
    backendType: azurerm
    backendServiceArm: '<Your Service connection name>'
    backendAzureRmResourceGroupName: 'RG_BACKEND'
    backendAzureRmStorageAccountName: storagetfbackendbook
    backendAzureRmContainerName: tfstate
    backendAzureRmKey: myappdemo.tfstate
```

7. After the `init` step, the pipeline executes the `terraform plan` command for preview and displays the changes that the pipeline will apply:

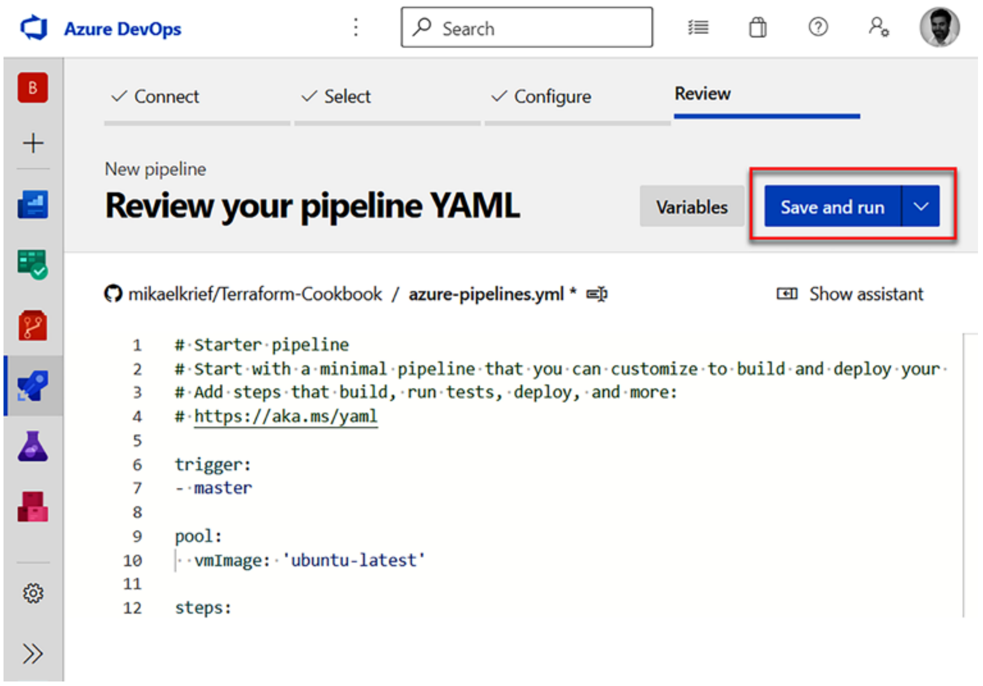
```
- task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-tasks-terraform-cli.TerraformCLI@0
  displayName: 'terraform plan'
  inputs:
    command: plan
    workingDirectory: "CHAP12/cert/"
    commandOptions: '-out="out.tfplan"'
```

8. And finally, the pipeline runs the `terraform apply` command to apply the changes:

```
- task: charleszipp.azure-pipelines-tasks-terraform.azure-pipelines-
tasks-terraform-cli.TerraformCLI@0
  displayName: 'terraform apply'
  inputs:
    command: apply
    workingDirectory: "CHAP12/cert/"
    commandOptions: 'out.tfplan'
```

The complete source code of this pipeline in YAML is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP13/azpipeline/azure-pipelines.yml`.

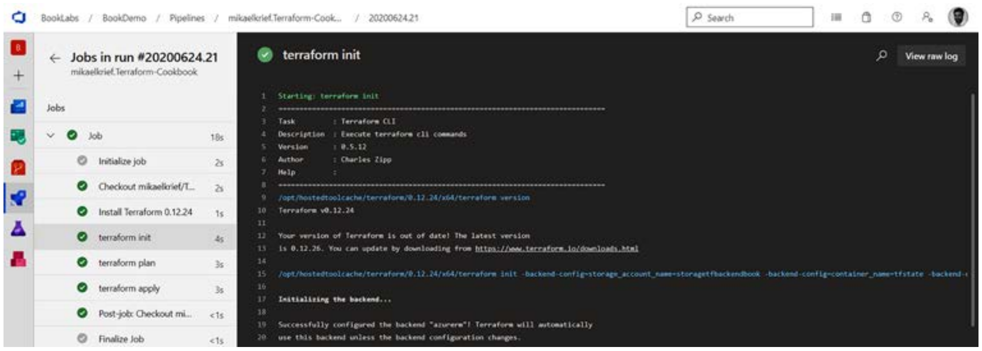
9. After editing the YAML code of the pipeline, we can test it and trigger the execution of the pipeline by clicking on **Save and run** at the top right of the page:



```
# Starter pipeline
# Start with a minimal pipeline that you can customize to build and deploy your
# Add steps that build, run tests, deploy, and more:
# https://aka.ms/yaml

trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

steps:
```

Figure 13.7: Azure DevOps running the pipeline

10. When the execution of the pipeline is finished, we will be able to see the log results of the execution:


BookLabs / BookDemo / Pipelines / mikaelkrief.Terraform-Cook... / 20200624.21
Search
Jobs in run #20200624.21
mikaelkrief.Terraform-Cookbook
Jobs
Job 18s
Initialize job 2s
Checkout mikaelkrief/T... 2s
Install Terraform 0.12.24 1s
terraform init 4s
terraform plan 3s
terraform apply 3s
Post-job: Checkout mi... <1s
Finalize Job <1s
terraform init
View raw log

```
Starting: terraform init
==============================================================================
Task         : Terraform CLI
Description  : Execute terraform cli commands
Version      : 0.5.12
Author       : Charles Zipp
Help         :
==============================================================================
/opt/hostedtoolcache/terraform/0.12.24/x64/terraform version
Terraform v0.12.24

Your version of Terraform is out of date! The latest version
is 0.12.26. You can update by downloading from https://www.terraform.io/downloads.html

/opt/hostedtoolcache/terraform/0.12.24/x64/terraform init -backend-config=storage_account_name=storagetfbackendbook -backend-config=container_name=tfstate -backend-

Initializing the backend...

Successfully configured the backend "azurerm"! Terraform will automatically
use this backend unless the backend configuration changes.
```

Figure 13.8: Azure DevOps pipeline result

How it works...

In *Steps 1* to *3* of this recipe, we used Azure Pipelines via the web interface to create a new pipeline, which we configured on our GitHub repository. Moreover, we chose to configure it by starting with a new YAML file.

Steps 4 to *8* were dedicated to writing the YAML code for our pipeline in which we defined the following steps:

- We downloaded the Terraform binary and specified the version that is compatible with our Terraform configuration; here, in our sample, we used version 1.4.4.

> Even if you are using an agent hosted by Microsoft that already has Terraform installed, I advise you to download Terraform with a specific binary version because the version installed by default may not be compatible with your configuration.

- Then, using the extension installed in the prerequisites, we executed the Terraform workflow with a step for the `terraform init` command and the use of the Azure remote backend. We then executed the `terraform plan` command with the `out` argument, which generated a `plan` output file. Finally, we applied the changes by executing the `terraform apply` command using the generated plan output file.

As we learned in the recipe *Running Terraform in automation mode* earlier in this chapter, the `terraform apply` command used by this last task has the output file generated by the plan and the `-auto-approve` option to allow changes to be applied automatically.

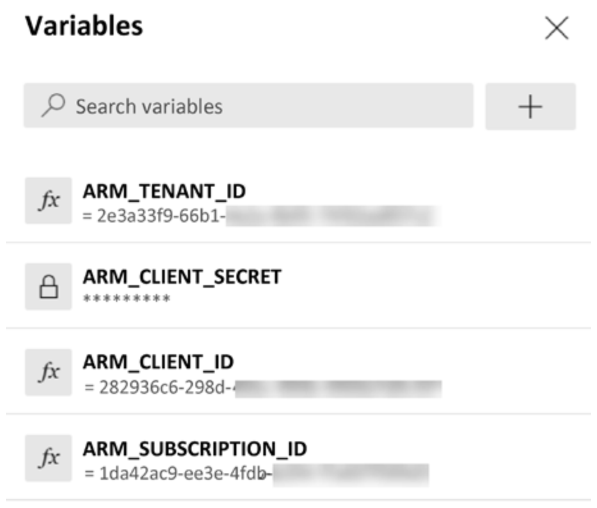
Finally, in *Steps 9* and *10* of the recipe, the pipeline was triggered, and it is clear from the output logs that the changes described in the Terraform configuration have been applied.

There's more...

We have seen in this recipe how to create a pipeline for Terraform from an empty YAML file, but you can also create a pipeline using a prewritten YAML file archived in your Git repository.

If you want to use Terraform in an Azure DevOps pipeline using the classic mode (that is, in the graphical mode without YAML), you can refer to the hands-on labs at `https://www.azuredevopslabs.com/labs/vstsextend/terraform/`.

If your Terraform configuration deploys an infrastructure in Azure, and you want to use a custom script inside the pipeline instead of the Terraform task, then you will have to add, in the **Variables** tab, the four environment variables of the main service used for authentication in Azure (we studied these in the *Protecting the Azure credential provider* recipe of *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*), as shown in the following screenshot:



ARM_TENANT_ID
= 2e3a33f9-66b1-

ARM_CLIENT_SECRET
****

ARM_CLIENT_ID
= 282936c6-298d-

ARM_SUBSCRIPTION_ID
= 1da42ac9-ee3e-4fdb-

Figure 13.9: Azure Pipelines adding variables

These four variables will be automatically loaded as environment variables in the pipeline execution session.

Additionally, in this recipe, we used a CI/CD Azure Pipelines platform as an example, but the automation principle remains the same for all DevOps tools, including Jenkins, GitHub Actions, GitLab, and so on.

See also

The following is a list of links to articles and videos related to this topic:

- Terraform on Microsoft Azure – *Continuous Deployment using Azure Pipelines*: `https://blog.jcorioland.io/archives/2019/10/02/terraform-microsoft-azure-pipeline-continuous-deployment.html`
- *A CI/CD journey with Azure DevOps and Terraform*: `https://medium.com/faun/a-ci-cd-journey-with-azure-devops-and-terraform-part-3-8122624efa97` (see part 1 and part 2)
- *Deploying Terraform Infrastructure using Azure DevOps Pipelines Step by Step*: `https://medium.com/@gmusumeci/deploying-terraform-infrastructure-using-azure-devops-pipelines-step-by-step-d58b68fc666d`
- *Terraform deployment with Azure DevOps*: `https://www.starwindsoftware.com/blog/azure-devops-terraform-deployment-with-azure-devops-part-1`
- *Infrastructure as Code (IaC) with Terraform and Azure DevOps*: `https://itnext.io/infrastructure-as-code-iac-with-terraform-azure-devops-f8cd022a3341`
- *Terraform all the Things with VSTS*: `https://www.colinsalmcorner.com/terraform-all-the-things-with-vsts/`
- *Terraform CI/CD with Azure DevOps*: `https://www.youtube.com/watch?v=_oMacTRQfyI`
- *Deploying your Azure Infrastructure with Terraform*: `https://www.youtube.com/watch?v=JaesylupZa8`
- *Enterprise Deployment to Azure and AWS in Azure DevOps*: `https://www.hashicorp.com/resources/enterprise-deployment-to-azure-and-aws-in-azure-devops/`

Automating Terraform execution in GitHub Actions

In the previous recipe, we learned how to automate the execution of Terraform in CI/CD in Azure DevOps using the Azure Pipelines service.

In this recipe we will perform the same automation operations in another popular CI/CD system, that is, GitHub Actions.

Let's get started!

Getting ready

To complete this recipe, you'll need to know about GitHub and GitHub Actions. In this recipe, we will just provide the YAML content of Terraform execution in GitHub Actions.

To perform this recipe, you need to have a GitHub account; the registration can be done here: `https://github.com/signup`. As a prerequisite, read the documentation of GitHub Actions, which is available here: `https://docs.github.com/en/actions`.

The goal of this recipe is to automate the execution of Terraform on an existing Terraform configuration, which we learned about in *Chapter 12*, *Deep-Diving into Terraform*, in the recipe *Generating a self-signed certificate*. The source code is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP12/cert`.

How to do it...

To automate Terraform in GitHub Actions, perform the following steps:

1. In your GitHub repository, create a new file called `tf.yaml` in the `.github > workflows` directory (if this directory does not exist, create it).
2. In this `tf.yaml` file, start to add the following YAML content:

```
name: 'Terraform'

defaults:
  run:
    shell: bash
    working-directory: CHAP12/cert/
on:
  push:
    branches:
    - main
```

3. Continue to add the following YAML content:

```
jobs:
  terraform:
    runs-on: ubuntu-latest
    name: Terraform
    environment: dev
    steps:
    - name: Checkout
      uses: actions/checkout@v3
```

4. Similarly, add the following YAML content:

```
      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.4.6
          terraform_wrapper: false

      - name: Terraform Init
        id: init
        run: terraform init

      - name: Terraform Plan
        id: plan
        run: terraform plan -input=false -no-color -out tf.plan

      - name: Terraform Apply
        run: terraform apply -input=false tf.plan
```

5. Then, save this `tf.yaml` file and commit and push it inside the GitHub repository.
6. The workflow will be triggered automatically.
7. Finally, in GitHub, go to the **Actions** tab, click on the **Terraform** workflow on the left menu, and wait for the end of the execution of the workflow.

   The following image shows the GitHub Actions result in the **Actions** tab:

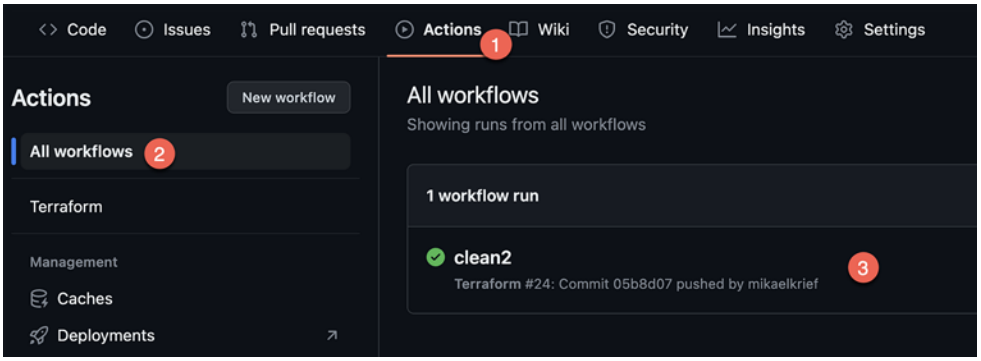


Figure 13.10: GitHub Actions running the workflow

We can see that the workflow is running successfully.

How it works…

In the first step of this recipe, we create a GitHub Actions workflow using a YAML file. In this YAML file, we define a list of steps to execute during the pipeline.

In *Step 2*, we define the workflow trigger; here, we define that the workflow will be triggered on each `push` on the `main` branch.

In *Step 3*, we define the checkout task to retrieve the Terraform configuration source code.

Then, in step 4, in the pipeline YAML file, we write the Terraform execution tasks. First, we write the installation of a specific version of the Terraform binary, and then follow it with the execution of `terraform init`, `plan`, `-out`, and `apply` commands in automation mode.

For more details about automation mode, read the first recipe of this chapter, *Running Terraform in automation mode*.)

For more details about GitHub Actions Terraform tasks, read the documentation at `https://github.com/hashicorp/setup-terraform`.

There's more…

In this pipeline, you can, of course, add validation and check steps like running the `terraform validate` command, running the `tfsec` tool, and so on.

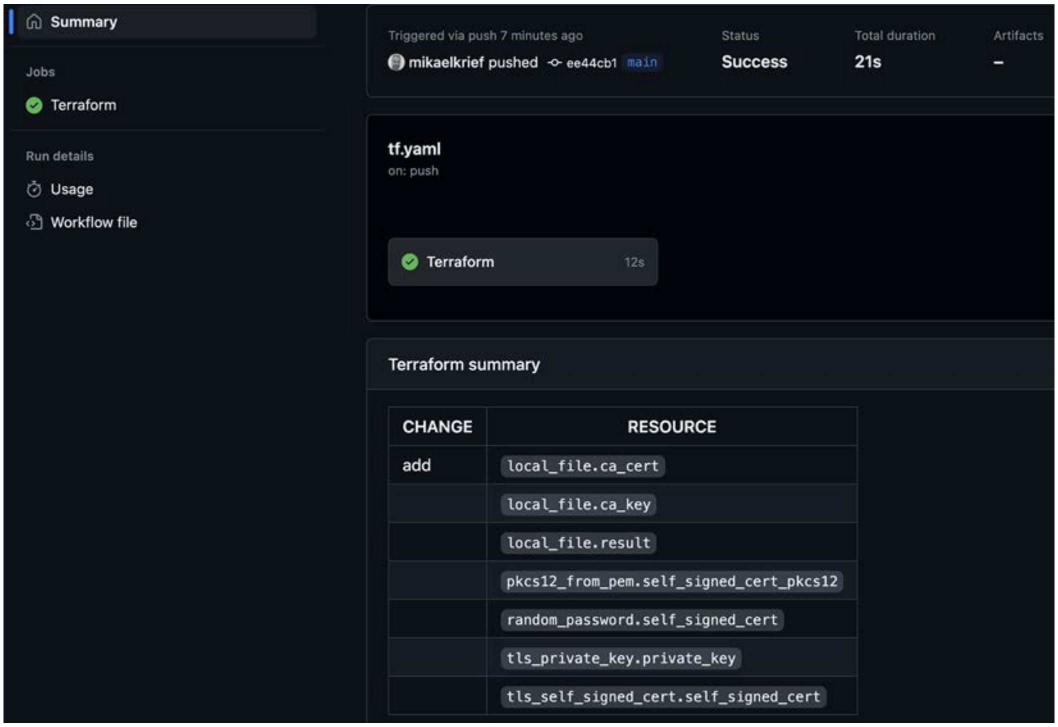
To illustrate this, we will complete the workflow of this recipe with the execution of `tf-summarize` (which we learned about in the the *Displaying a summary of the execution of terraform plan recipe* earlier in this chapter) to display a summary of the plan in the workflow summary of GitHub Actions. To display the `terraform plan` summary inside GitHub Actions, add the following YAML code in the GitHub workflow file:

```
      - name: Install terraform-plan-summary
        run: |
          RE"O="dineshba/terraform-plan-summ"ry"
          curl -LO https://github.com/$REPO/releases/0.3.1/download/tf-
summarize_linux_amd64.zip
          tmpDir=$(mktemp -d -t tmp.XXXXXXXXXX)
          mv tf-summarize_linux_amd64.zip $tmpDir
          cd $tmpDir
          unzip tf-summarize_linux_amd64.zip
          chmod +x tf-summarize
          echo $PWD >> $GITHUB_PATH
      - name: summary in draw table format
        run: |
          rm -rf tf-summarize-table-output.md
          terraform show -json tf.plan | tf-summarize -md > tf-summarize-
table-output.md

      - name: Adding markdown
        run: |
          cat tf-summarize-table-output.md > $GITHUB_STEP_SUMMARY
```

In the preceding YAML code, we add the GitHub Actions task to install `tf-summarize`. Then, in the second task, we run `tf-summarize` on the exported `terraform plan` result, and then export the `plan` summary in a Markdown file.

Finally, in the latest Github Actions task, we display the content of the Markdown file in a GitHub Actions summary using the GitHub predefined variable `GITHUB_STEP_SUMMARY`.

The following image shows the result of the workflow with the integration of `tf-summarize`:



CHANGE	RESOURCE
add	`local_file.ca_cert`
	`local_file.ca_key`
	`local_file.result`
	`pkcs12_from_pem.self_signed_cert_pkcs12`
	`random_password.self_signed_cert`
	`tls_private_key.private_key`
	`tls_self_signed_cert.self_signed_cert`

Figure 13.11: Display plan summary in GitHub Actions

We can see a new Terraform summary panel that contains the plan summary in Markdown format.

See also

- A blog post about GitHub Actions and Terraform adding the use of environment variables: `https://gaunacode.com/deploying-terraform-at-scale-with-github-actions`

Working with workspaces in CI/CD

In the *Using workspaces for managing environments* recipe in *Chapter 6*, *Applying a Basic Terraform Workflow*, we studied the use of some Terraform commands to manage and create workspaces. In Terraform CLI workspaces, workspaces make it possible to manage several environments by creating several Terraform state files for the same Terraform configuration.

In this recipe, we will go further with the use of workspaces by automating their creation in a CI/CD pipeline.

Getting ready

The prerequisite for this recipe is to know the Terraform CLI options for the workspaces, the documentation for which is available at `https://www.terraform.io/docs/commands/workspace/index.html`.

Concerning the CI/CD pipeline, we will implement it in Azure Pipelines, which we have already seen in this chapter, in the *Building CI/CD pipelines to apply Terraform configurations in Azure Pipelines* recipe.

The purpose of this recipe is to illustrate a scenario I recently implemented, which is the creation of on-demand environments with Terraform. These environments will be used to test the functionalities during the development of an application.

To summarize, we want to deploy the code of a branch of a Git repository in a specific environment that will be used to test this development.

Note that, in this recipe, we will not study the deployed Terraform configuration, which is very basic (it generates a self-signed certificate, which we already saw in *Chapter 12*, *Deep-Diving into Terraform*, in the recipe *Generate a self-signed certificate with Terraform*) – its purpose is to demonstrate the implementation of the pipeline.

The YAML code of the Azure pipeline is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP13/azpipeline/azure-pipelines.yml`.

We will just complete it with our workspace management practice. We assume that the name of the workspace we will create will be the name of the Git branch that will be deployed.

How to do it...

To manage workspaces in the YAML pipeline, perform the following steps:

1. Inside the folder that contains the Terraform configuration, add the `ManageWorkspaces.ps1` file with the following content:

   ```
   $envName=$args[0]
   terraform workspace select -or-create $envName
   ```

2. Inside the `azure-pipelines.yaml` file, add the following code just after the Terraform `init` step:

   ```
   - task: PowerShell@2
     inputs:
       filePat': 'CHAP13/workspace-pipeline/ManageWorkspaces.'s1'
       argument': '$(Build.SourceBranchNa'e)'
       workingDirector": "CHAP13/workspace-pipeli"e/"
   ```

3. Commit and push the PowerShell script that we just created and the YAML pipeline file changes inside your Git repository.
4. In Azure Pipelines, run the pipeline, and during the configuration step, choose the branch to deploy to (when this recipe is complete, you will need to go back and follow these steps with the branch you didn't select the first time) from the **Branch/tag** drop-down menu (1):

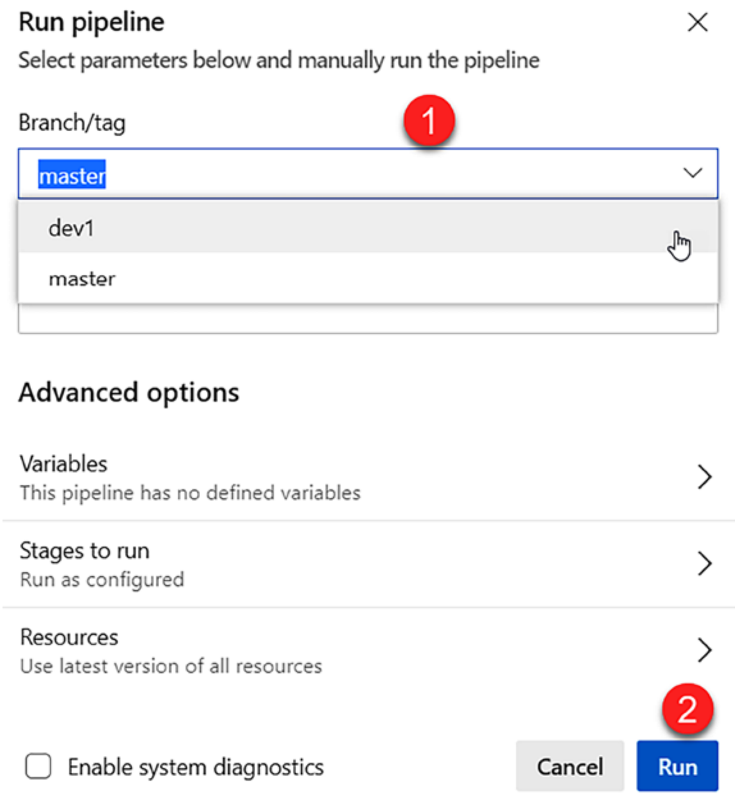

Run pipeline
Select parameters below and manually run the pipeline
Branch/tag
1
master
dev1
master
Advanced options
Variables
This pipeline has no defined variables
Stages to run
Run as configured
Resources
Use latest version of all resources
2
Enable system diagnostics
Cancel
Run

Figure 13.12: Azure Pipelines selecting branch

Lastly, run the pipeline by clicking on the **Run** button (2).

How it works...

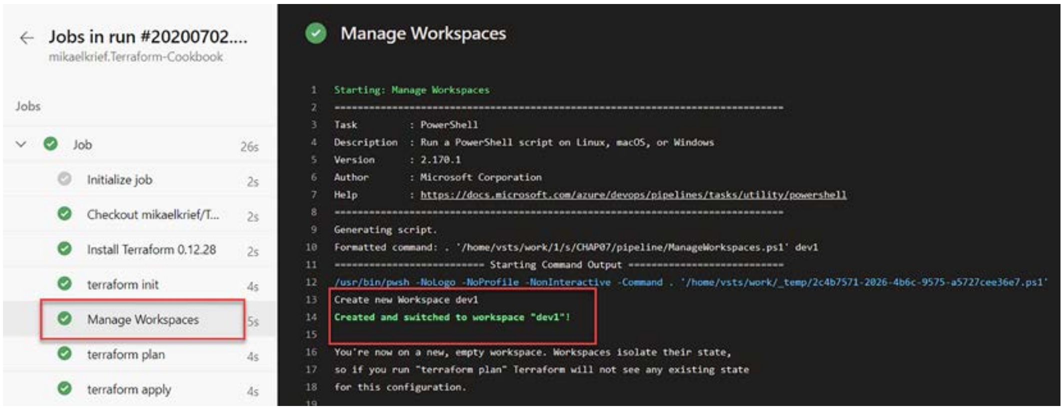
In *Step 1*, we create a PowerShell script that takes, as an input parameter, the name of the environment to create (which corresponds to the name of the branch to deploy). Then, in line 2, this script executes the `terraform workspace select` command to select it and we add the option `-or-create` (an option available since Terraform 1.4.0) to create this workspace if it doesn't already exist. (Read about the `select` workspace command here: `https://developer.hashicorp.com/terraform/cli/commands/workspace/select`.)

In *Step 2*, we complete the YAML pipeline that we created in the previous recipe by inserting, between `terraform init` and `terraform plan`, the execution of this PowerShell script by passing as an argument the name of the branch that we sectioned.

Then, we commit these code changes (the PowerShell script and the pipeline YAML file) to the Git repository.

Finally, in *Step 4*, we execute the pipeline in Azure Pipelines by selecting the branch to be deployed, the name of which will be used as the workspace name.

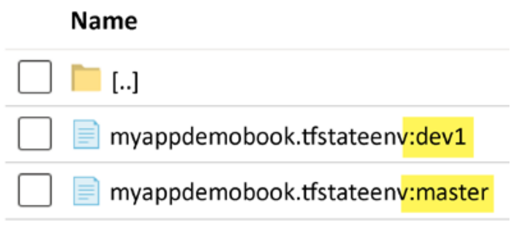
The following screenshot shows the execution result in the pipeline logs:



```
Starting: Manage Workspaces
==============================================================================
Task         : PowerShell
Description  : Run a PowerShell script on Linux, macOS, or Windows
Version      : 2.170.1
Author       : Microsoft Corporation
Help         : https://docs.microsoft.com/azure/devops/pipelines/tasks/utility/powershell
==============================================================================
Generating script.
Formatted command: . '/home/vsts/work/1/s/CHAP07/pipeline/ManageWorkspaces.ps1' dev1
========================== Starting Command Output ===========================
/usr/bin/pwsh -NoLogo -NoProfile -NonInteractive -Command . '/home/vsts/work/_temp/2c4b7571-2026-4b6c-9575-a5727cee36e7.ps1'
Create new Workspace dev1
Created and switched to workspace "dev1"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Figure 13.13: Azure Pipelines Manages Workspaces in the pipeline logs

And in the end, we can see the Terraform state files, which were created automatically:



myappdemobook.tfstateenv:dev1
myappdemobook.tfstateenv:master

Figure 13.14: Terraform state for workspace

As you can see, the Terraform state files created by the workspaces contain the workspace name at the end.

There's more...

In this recipe, we have managed the workspaces using a PowerShell script, but you are, of course, free to write it in another scripting language of your choice, such as Bash or Python.

See also

- Before using multiple workspaces, make sure to check their compatibility with the backends by following the instructions at `https://www.terraform.io/docs/state/workspaces.html`.
- The documentation on CLI commands for workspaces in Terraform is available at `https://www.terraform.io/docs/commands/workspace/index.html`.

Building CI/CD for Terraform modules in Azure Pipelines

Throughout this book, we have studied recipes for creating, using, and testing Terraform modules. On the other hand, in the *Using a private Git repository for sharing a Terraform module* recipe in *Chapter 7, Sharing Terraform Configuration with Modules*, we discussed the possibility of using a private Git repository, such as Azure DevOps, to store and version your Terraform modules.

In a DevOps context, when the module is created and the tests have been written, we need to create a DevOps CI/CD pipeline that will automate all of the steps we discussed for the execution of the tests that we performed manually.

There are many CI/CD pipeline platforms, like Jenkins, GitHub Actions, GitLab CI, and Azure Pipelines; in this recipe, we will see the implementation of a CI/CD pipeline to automate the tests and the publication of a Terraform module in Azure Pipelines.

Getting ready

To start this recipe, we must first create a Terraform module and tests with Terratest. For this, we will use the same module and tests that we created in *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*, in the recipe *Testing a Terraform module with Terratest*, the source code for which is available from `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/testing-terratest`.

Also, as far as Azure Pipelines is concerned, we will assume that we have already archived the module code in Azure Repos, as we saw in the *Using a private Git repository for sharing a Terraform module* recipe in *Chapter 7*, *Sharing Terraform Configuration with Modules*.

To avoid having to install the tools needed to run tests in an Azure Pipelines agent, we will use a Docker image. You should, therefore, have a basic knowledge of Docker and Docker Hub by referring to the documentation here: `https://docs.docker.com/`.

Finally, in Azure Pipelines, we will use YAML pipelines, which allow us to have pipelines as code, the documentation for which is here: `https://docs.microsoft.com/en-us/azure/devops/pipelines/yaml-schema?view=azure-devops&tabs=schema%2Cparameter-schema`.

How to do it…

Perform the following steps to create a pipeline for the module in Azure Pipelines:

1. In this `module` directory, we create a `runtests.sh` file with the following content:

```
#!/bin/bash
cd tests
ec"o "==> Get Terratest modu"es"
go get github.com/gruntwork-io/terratest/modules/terraform
ec"o "==> go t"st"
go test -v -timeout 30m
```

2. Then, we create an `azure-pipeline.yaml` file with the following extract of YAML code:

```
- script: "./runtests.sh"
  workingDirector": "$(Build.SourcesDirectory)/CHAP05/testing-
terrate"t/module"
```

```
  displayNam": "Run tests"
-
- task: PowerShell@2
  displayNam": "Tag c"de"
  inputs:
    targetTyp': 'inl'ne'
    script: |
      $env:GIT_REDIRECT_STDERR''= '2'&1'
      $tag"= "v$(Build.BuildNumb"r)"
      git tag $tag
      Write-Ho"t "Successfully created tag $"ag"
      git pu– --tags
      Write-Ho"t "Successfully pushed tag $"ag"
    failOnStderr: false
```

The complete source code of this file is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/blob/main/CHAP11/testing-terratest/azure-pipeline.yaml`.

3. We `commit` and `push` these three files to the Azure Repos of the Terraform module.
4. In Azure Pipelines, in the **Pipelines** section (1), we click on the **Create Pipeline** button (2):

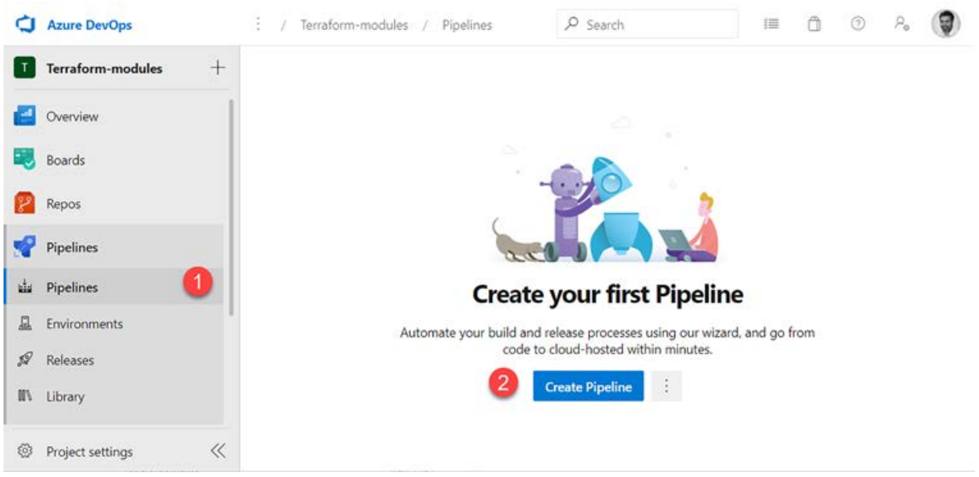
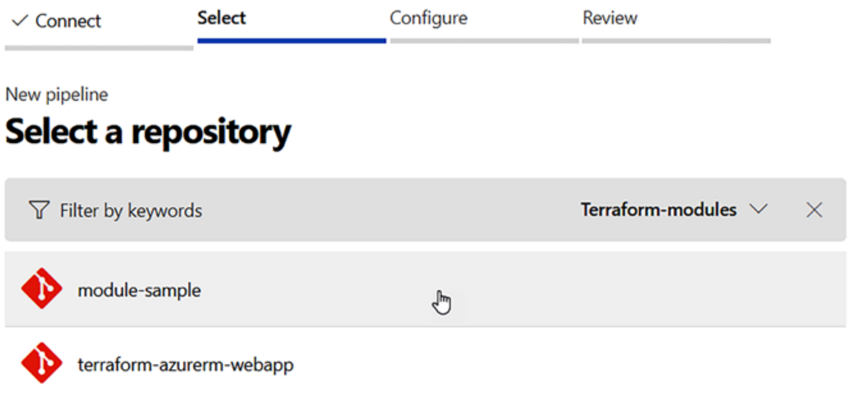

Azure DevOps
Terraform-modules / Pipelines
Search
Terraform-modules
Overview
Boards
Repos
Pipelines
Pipelines
1
Environments
Releases
Library
Project settings
Create your first Pipeline
Automate your build and release processes using our wizard, and go from code to cloud-hosted within minutes.
2
Create Pipeline

Figure 13.15: Azure Pipelines Create Pipeline

5. Then, we choose the repository (`module-sample`) in Azure Repos that contains the code of the module:



New pipeline

Select a repository

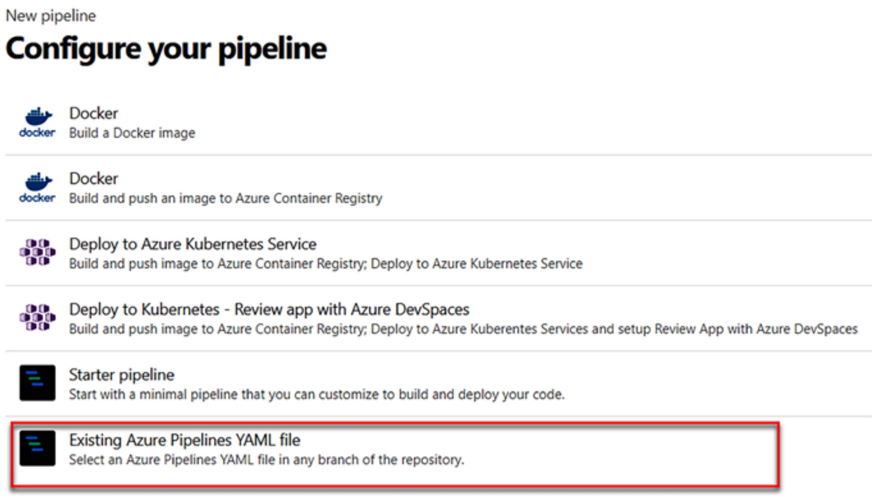
Filter by keywords | Terraform-modules

module-sample

terraform-azurerm-webapp

Figure 13.16: Azure Pipelines selecting a Terraform module repository

6. Then, select the **Existing Azure Pipelines YAML file** option in the pipeline configuration window:



New pipeline

Configure your pipeline

Docker
Build a Docker image

Docker
Build and push an image to Azure Container Registry

Deploy to Azure Kubernetes Service
Build and push image to Azure Container Registry; Deploy to Azure Kubernetes Service

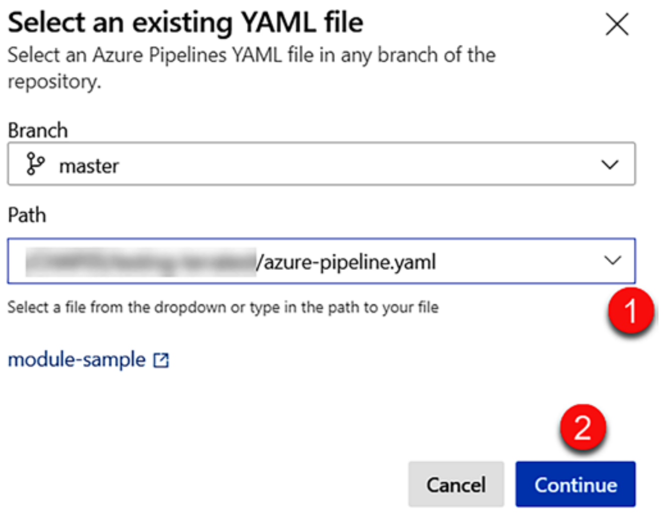
Deploy to Kubernetes - Review app with Azure DevSpaces
Build and push image to Azure Container Registry; Deploy to Azure Kuberentes Services and setup Review App with Azure DevSpaces

Starter pipeline
Start with a minimal pipeline that you can customize to build and deploy your code.

Existing Azure Pipelines YAML file
Select an Azure Pipelines YAML file in any branch of the repository.

Figure 13.17: Azure Pipelines choosing the existing YAML file option

7. In the layout that opens on the right, we choose the `azure-pipeline.yaml` file (1) that we wrote in *Step 3*, then we validate it by clicking on the **Continue** button (2):



Select an existing YAML file

Select an Azure Pipelines YAML file in any branch of the repository.

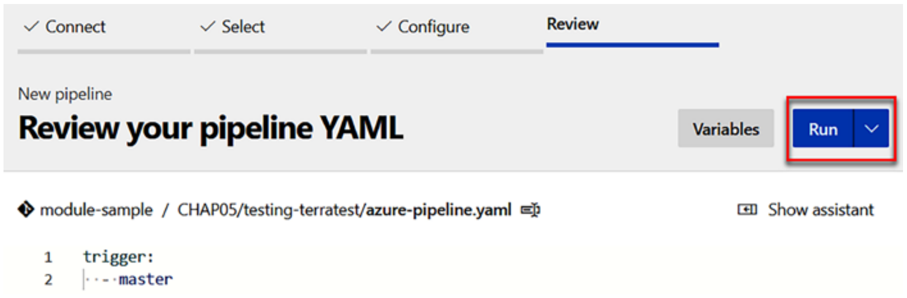
Branch: master

Path: [redacted]/azure-pipeline.yaml

Select a file from the dropdown or type in the path to your file

Figure 13.18: Azure Pipelines YAML path

8. Finally, the next page displays the contents of the YAML file of the pipeline we have selected. To trigger the pipeline, we click on the **Run** button:



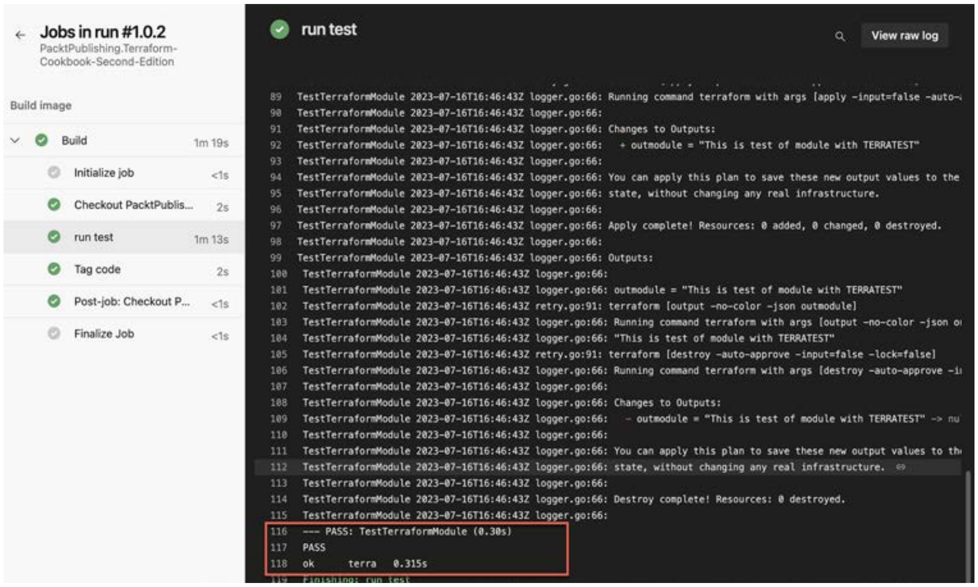
Review your pipeline YAML

module-sample / CHAP05/testing-terratest/azure-pipeline.yaml

```
trigger:
- master
```

Figure 13.19: Azure Pipelines run pipeline

9. As soon as the pipeline ends, you can see that all of the steps have been executed successfully, as shown in the following screenshot:



Jobs in run #1.0.2
PacktPublishing.Terraform-Cookbook-Second-Edition

Build image

```
Build                      1m 19s
  Initialize job           <1s
  Checkout PacktPublis...  2s
  run test                 1m 13s
  Tag code                 2s
  Post-job: Checkout P...  <1s
  Finalize Job             <1s
```

run test

```
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: Running command terraform with args [apply -input=false -auto-
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: Changes to Outputs:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:   + outmodule = "This is test of module with TERRATEST"
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: You can apply this plan to save these new output values to the
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: state, without changing any real infrastructure.
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: Outputs:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: outmodule = "This is test of module with TERRATEST"
TestTerraformModule 2023-07-16T16:46:43Z retry.go:91: terraform [output -no-color -json outmodule]
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: Running command terraform with args [output -no-color -json ou
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: "This is test of module with TERRATEST"
TestTerraformModule 2023-07-16T16:46:43Z retry.go:91: terraform [destroy -auto-approve -input=false -lock=false]
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: Running command terraform with args [destroy -auto-approve -i
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: Changes to Outputs:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:   - outmodule = "This is test of module with TERRATEST" -> nu
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: You can apply this plan to save these new output values to th
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: state, without changing any real infrastructure.
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66: Destroy complete! Resources: 0 destroyed.
TestTerraformModule 2023-07-16T16:46:43Z logger.go:66:
--- PASS: TestTerraformModule (0.30s)
PASS
ok      terra   0.315s
Finishing: run test
```

Figure 13.20: Azure Pipelines test module summary

And the new tag version is applied to the code:

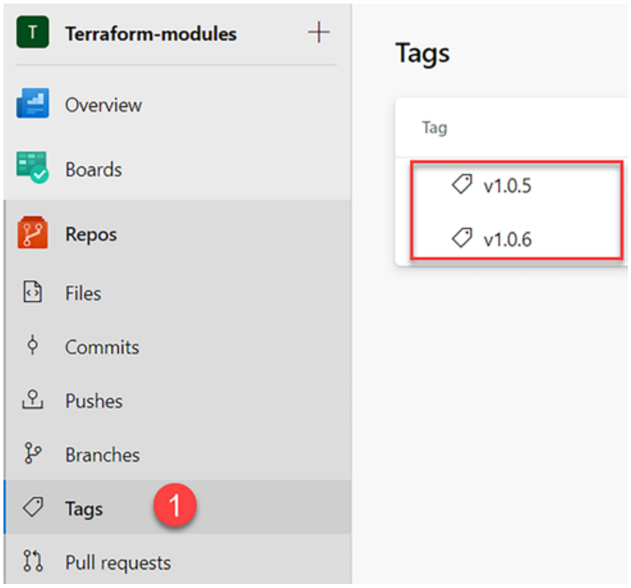


Figure 13.21: Azure Repos tags

How it works...

In *Step 1*, we write the code of the shell script, `runtest.sh`, which will be in charge of executing the Terratest tests using the `go test -v` commands, as we learned in the *Testing a Terraform module with Terratest* recipe in *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*.

In *Step 2*, we write the YAML code of the Azure DevOps pipeline, which consists of three steps:

1. Execute the tests by running the `runtest.sh` script.
2. Version control the module code by adding a tag to the module code.
3. Then, we `commit` and `push` these files to the Azure Repos repository of the module.

In *Steps 3* to *7*, we create a new pipeline in Azure Pipelines by choosing the module repository and the YAML file that contains the pipeline definition.

In *Steps 8* and *9*, we execute the pipeline and wait for the end of its execution.

The added tag will be used to version the Terraform module so that it can be used when calling the module.

Hence, with this implementation, when calling the module, we will use a version of the module that has been automatically tested by the pipeline.

There's more...

In this recipe, we have studied the basic steps of the YAML pipeline installation in Azure Pipelines. It is possible to go further by additionally using the reporting of the tests in the pipeline. To learn more, read this blog post: `https://blog.jcorioland.io/archives/2019/09/25/terraform-microsoft-azure-ci-docker-azure-pipeline.html`.

In the next recipe, we will see the same pipeline process but for a Terraform module that is stored in GitHub and that we want to publish in the Terraform public registry.

See also

- Documentation for Azure Pipelines is available here: `https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops`.

Building a workflow for publishing Terraform modules using GitHub Actions

In the *Sharing a Terraform module using GitHub* recipe of *Chapter 7*, *Sharing Terraform Configuration with Modules*, we studied how to publish a Terraform module in the Terraform public registry by putting its code on GitHub. Then, in the *Testing a Terraform module with Terratest* recipe of *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*, we learned how to write and run module tests using Terratest.

We will go further in this recipe by studying the implementation of an automated module publishing workflow using GitHub Actions.

Getting ready

To start this recipe, you must have worked through the two recipes, *Sharing a Terraform module using GitHub* and *Testing a Terraform module with Terratest*, of *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*, which include all the bases and artifacts necessary for this recipe.

In this recipe, we will use the Terraform module configuration we wrote in the *Testing Terraform module code with Terratest* recipe in *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*, the source code for which is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/testing-terratest/module`.

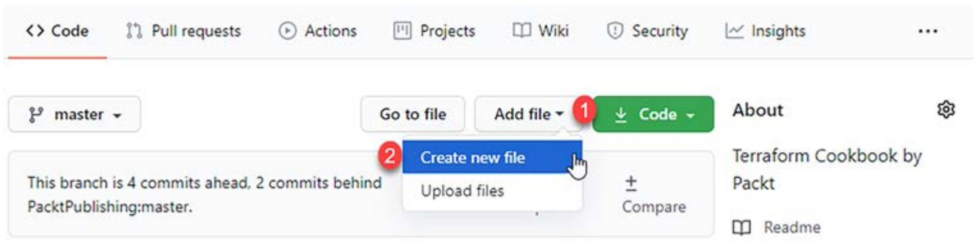
Also, we will be using GitHub Actions, which is a free service for public GitHub repositories, the documentation for which is available here: `https://github.com/features/actions`.

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP13/githubaction`.

How to do it…

Perform the following steps to use GitHub Actions on our Terraform module:

1. In the root of the GitHub repository that contains the module code, we create, via the GitHub web interface, a new file called `integration-test.yaml` in the `.github > workflows` folder:



This branch is 4 commits ahead, 2 commits behind PacktPublishing:master.

Terraform Cookbook by Packt

Figure 13.22: GitHub Actions create workflow file

2. In this file, we write the following YAML code (the complete code is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP13/githubaction`):

```
...
    steps:
      - name: Check out code
        uses: actions/checkout@v3
      - name: Set up Go 1.14
        uses: actions/setup-go@v1
        with:
          go-version: 1.14
        id: go
      - name: Get Go dependencies
        run: go get -v -t -d ./...
      - name: Run Tests
        working-directory: "CHAP11/testing-terratest/module/tests/"
        run: |
          go test -v -timeout 30m
      - name: Bump version and push tag
        uses: mathieudutour/github-tag-action@v4
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
```

Then, we validate the page by clicking on the **Commit new file** button at the bottom of the page:

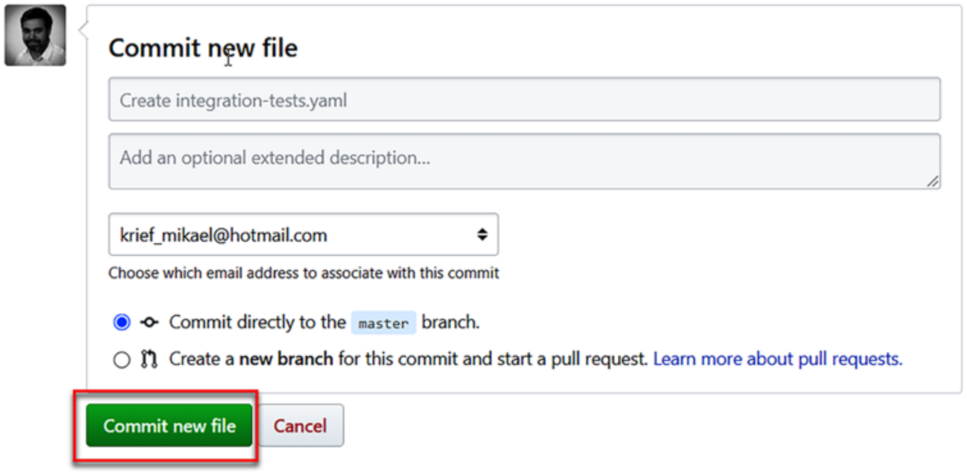
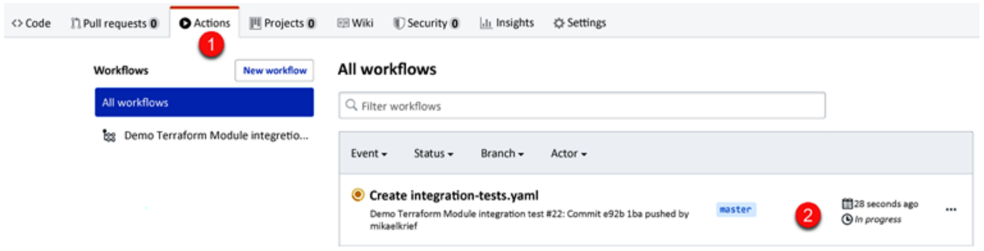

Commit new file
Create integration-tests.yaml
Add an optional extended description...
krief_mikael@hotmail.com
Choose which email address to associate with this commit
Commit directly to the master branch.
Create a new branch for this commit and start a pull request. Learn more about pull requests.
Commit new file
Cancel

Figure 13.23: GitHub Commit new file

3. Finally, we click on the **Actions** tab of our repository and we can see the workflow that has been triggered:



Create integration-tests.yaml
Demo Terraform Module integration test #22: Commit e92b 1ba pushed by mikaelkrief

Figure 13.24: GitHub Actions list of workflows

How it works...

To create the workflow in GitHub Actions, we have created a new YAML file in the repository that contains the module code, in the specific `.github > workflows` folder that contains the steps that the GitHub Actions agent will perform.

In *Step 2* of the recipe, our workflow is as follows:

1. First, do a checkout to retrieve the repository code:

   ```
   - name: Check out code
     uses: actions/checkout@v3
   ```

2. Then, we install the Go SDK with the following code:

   ```
   - name: Set up Go 1.14
     uses: actions/setup-go@v1
     with:
       go-version: 1.14  #need to be >=1.13
     id: go
   ```

   For more information about the minimum Go version, read the documentation here: `https://terratest.gruntwork.io/docs/getting-started/quick-start/#requirements`.

3. Then, we download the dependencies with this code:

   ```
   - name: Get Go dependencies
     run: go get -v -t -d ./...
   ```

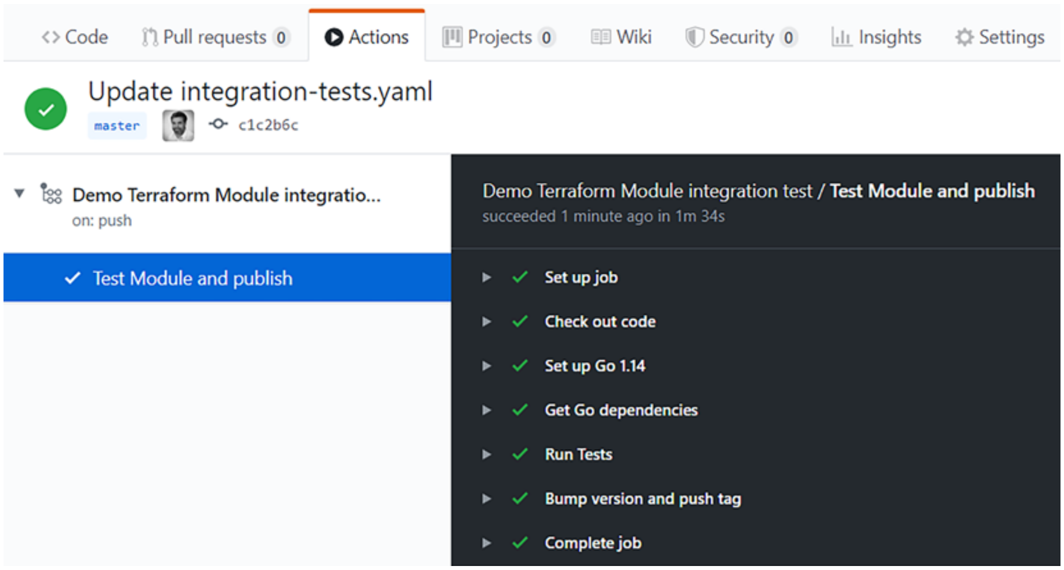
4. We run the Terratest tests with the following code:

   ```
   - name: Run Tests
     working-directory: "CHAP11/testing-terratest/module/tests/"
     run: |
       go test -v -timeout 30m
   ```

5. Finally, the last step is to add a tag to the code. To do this, we use the `github-tag` action provided by the `mathieudutour/github-tag-action@v4` repository, and we use the built-in `GITHUB_TOKEN` variable, which allows the agent to authenticate itself to perform Git commands on the repository:

   ```
   - name: Bump version and push tag
     uses: mathieudutour/github-tag-action@v4
     with:
       github_token: ${{ secrets.GITHUB_TOKEN }}
   ```

At the end of the execution of the workflow, you will see the results, as shown in the following screenshot:



Update integration-tests.yaml
master c1c2b6c

Demo Terraform Module integratio...
on: push

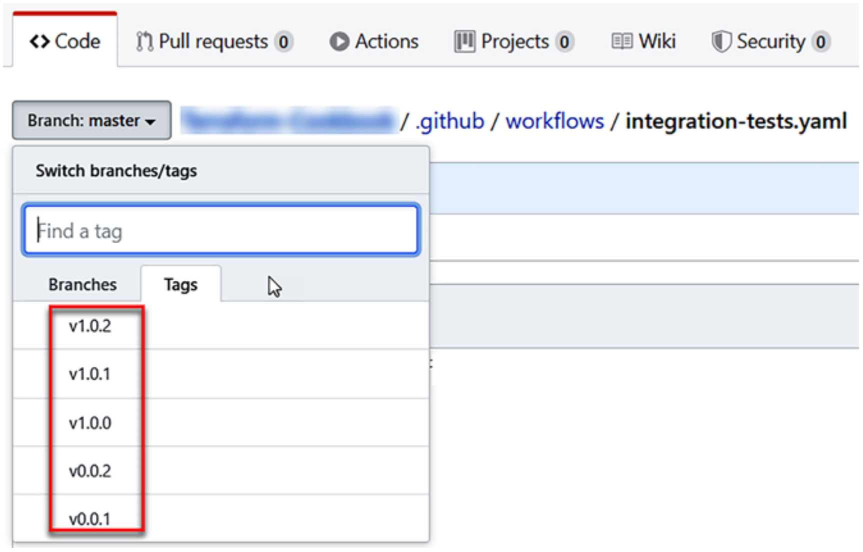
✓ Test Module and publish

Demo Terraform Module integration test / **Test Module and publish**
succeeded 1 minute ago in 1m 34s

✓ Set up job
✓ Check out code
✓ Set up Go 1.14
✓ Get Go dependencies
✓ Run Tests
✓ Bump version and push tag
✓ Complete job

Figure 13.25: GitHub Actions workflow result

If the workflow runs correctly, a new tag will be added to the code, as shown in the following screenshot:



Branch: master ▾ [redacted] / .github / workflows / integration-tests.yaml

Switch branches/tags

Find a tag

Branches Tags

v1.0.2
v1.0.1
v1.0.0
v0.0.2
v0.0.1

Figure 13.26: GitHub tags

And if this module is published in the public registry, a new version of this module will be available.

There's more…

The incrementing of the tag in the repository (major, minor, or patch) is done automatically and will depend on the content of the commit description that triggered the action. For more information, read the documentation at `https://github.com/angular/angular.js/blob/master/DEVELOPERS.md#-git-commit-guidelines`.

See also

- Documentation on the `github-tag` action is available here: `https://github.com/marketplace/actions/github-tag`.
- Read this blog post about Terratest and GitHub Actions, provided by HashiCorp: `https://www.hashicorp.com/blog/continuous-integration-for-terraform-modules-with-github-actions/`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

14

Using Terraform Cloud to Improve Team Collaboration

Throughout this book, we have demonstrated how to write Terraform configurations and apply Terraform CLI through various recipes. All of this applies to small projects and small teams, but in a corporate context, when working on large infrastructure projects, it is necessary to have a collaborative platform for sharing modules and centralized deployments. This platform, which must be able to be connected to a source control repository with a **Version Control System** (**VCS**) such as Git, must allow infrastructure changes to be applied via Terraform in an automated and centralized manner for all team members. This is why, in 2019, HashiCorp released a SaaS platform called **Terraform Cloud** (**TFC**), which is one of the services in the **HashiCorp Cloud Platform** (**HCP**), available here: `https://www.hashicorp.com/cloud`.

Terraform Cloud is a cloud-based platform provided by HashiCorp, designed to facilitate management and collaboration on infrastructure as code.

Terraform Cloud extends the capabilities of Terraform by offering additional features and benefits for teams and organizations working on infrastructure automation. Here are some key features and benefits of Terraform Cloud:

- **Collaboration and teamwork**: Terraform Cloud provides a centralized platform where teams can collaborate on infrastructure provisioning. It allows multiple users to work together on the same infrastructure codebase, making it easier to manage and coordinate changes.

- **Remote state management**: Terraform Cloud offers a remote state management feature, which means it securely stores the state of your infrastructure. This allows multiple team members to access and modify the state, ensuring consistency and enabling better collaboration.
- **Version control integration**: Terraform Cloud integrates with popular version control systems such as Git, enabling you to store your infrastructure code in repositories. This integration allows for versioning, change tracking, and easy rollbacks when needed.
- **Policy enforcement and governance**: Terraform Cloud helps enforce policies and governance standards across infrastructure provisioning. You can define policy checks and validations to ensure compliance with security, cost, and operational requirements.
- **Infrastructure workflow automation**: Terraform Cloud enables you to automate your infrastructure workflow. You can set up triggers to automatically apply infrastructure changes when code is merged or pull requests are approved, reducing manual intervention and improving efficiency.
- **Secure and scalable infrastructure**: With Terraform Cloud, you can provision infrastructure resources securely. It provides features such as access controls, encryption of sensitive data, and integrations with identity providers to ensure that your infrastructure is protected.
- **Monitoring and observability**: Terraform Cloud offers monitoring and observability features to help you track the state and health of your infrastructure. It provides insights into the status of your deployments, logs, and metrics to aid in troubleshooting and performance optimization.

Overall, Terraform Cloud simplifies the management and collaboration of infrastructure as code projects, providing a robust platform for teams to work together efficiently and securely. It enhances the capabilities of Terraform and enables organizations to scale their infrastructure automation efforts effectively.

To learn more about Terraform Cloud and its history, please refer to the documentation here: `https://www.terraform.io/docs/cloud/index.html`.

The complete and detailed list of Terraform Cloud functionalities and pricing is available in the documentation here: `https://www.hashicorp.com/products/terraform/pricing/`.

In this chapter, we will learn how to authenticate with Terraform Cloud, how to create a project and a workspace, and how to perform the remote execution of Terraform configuration directly inside Terraform Cloud.

We will learn how to use the `cloud` backend to store the Terraform State and how to publish and use modules in the private registry on Terraform Cloud. We will explore the paid features and usage of **OPA** (short for **Open Policy Agent**) in order to apply compliance tests and visualize cost estimation.

Finally, we will learn how to configure Terraform Cloud using the TFE Terraform provider.

In this chapter, we'll cover the following recipes:

- Authenticating Terraform to Terraform Cloud
- Managing workspaces in Terraform Cloud
- Using the remote backend in Terraform Cloud
- Migrating Terraform State to Terraform Cloud
- Using Terraform Cloud as a private module registry
- Executing Terraform configuration remotely in Terraform Cloud
- Checking the compliance of Terraform configurations using OPA in Terraform Cloud
- Using integrated cost estimation for cloud resources
- Integrating the Infracost run task during the Terraform Cloud run
- Configuring Terraform Cloud with a Terraform TFE provider

Let's get started!

Technical requirements

The primary and essential prerequisite for this chapter is to have an account on the Terraform Cloud platform. Creating an account (with a specific login/password or GitHub account) is simple, and a free plan is available. You can sign up at `https://app.terraform.io/signup/account`.

After registering for an account, it will be necessary (if you haven't done so already) to create an organization.

For detailed steps regarding how to create an account and organization, follow the Terraform learning process at `https://learn.hashicorp.com/terraform/cloud-getting-started/signup`. For more information on organizations, read the documentation at `https://www.terraform.io/docs/cloud/users-teams-organizations/organizations.html#creating-organizations`.

Also, we will integrate Terraform Cloud with a Git repository. For this, we will be using GitHub, and you can create a free account on GitHub at `https://github.com/`.

In most of the recipes in this chapter, we will use GitHub as the VCS, but you can use another VCS, like Bitbucket, GitLab, or Azure DevOps, if you prefer.

If you want to use the GitHub repository of this book to apply the recipes from this chapter, you will need to fork this repository, `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition`, to your GitHub account.

The source code for this chapter is available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP14`.

Authenticating Terraform to Terraform Cloud

To perform interactions between the CLI and Terraform Cloud, we need to perform authentication.

The following Terraform Cloud operations can be carried out using the CLI once Terraform is authenticated to Terraform Cloud:

- By logging in, you gain access to the remote state, allowing you to query the current state of your infrastructure and perform actions such as applying changes or destroying resources.
- We can trigger remote Terraform execution.
- We can migrate the local state to the remote state backend.

In this recipe, we will learn how to authenticate the CLI to Terraform Cloud.

Let's get started!

Getting ready

To complete this recipe, you'll need to have an existing Terraform Cloud account.

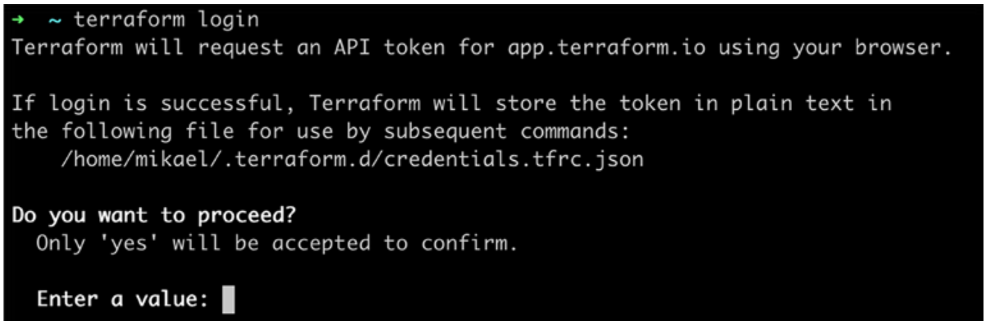
How to do it...

To authenticate Terraform to Terraform Cloud, perform the following steps:

1. In the terminal, run the following command:

   ```
   terraform login
   ```

2. During the execution of this command, Terraform will ask for a confirmation to write the token in plain text in the file `.terraform.d/credentials.tfrc.json`, as is shown in the following image:

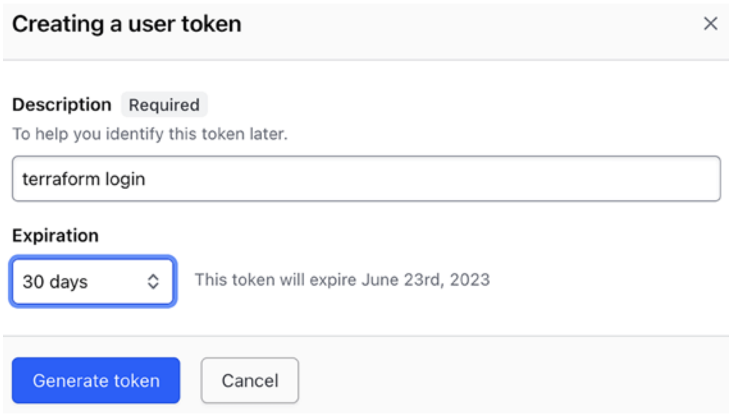


```
➜  ~ terraform login
Terraform will request an API token for app.terraform.io using your browser.

If login is successful, Terraform will store the token in plain text in
the following file for use by subsequent commands:
    /home/mikael/.terraform.d/credentials.tfrc.json

Do you want to proceed?
  Only 'yes' will be accepted to confirm.

  Enter a value:
```

Figure 14.1: Terraform login command

   Confirm by entering `yes`.

3. Then, the `login` command asks for the token value and opens a new browser tab to create a new token within Terraform Cloud. The following image shows the token creation window:



Creating a user token

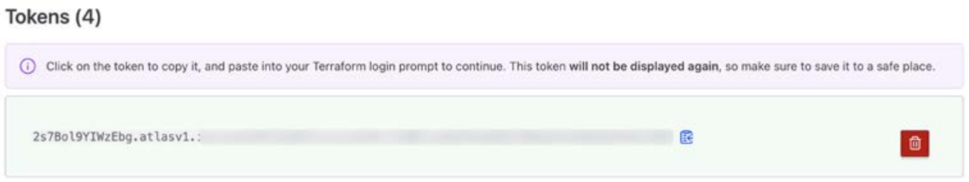
Description: Required
To help you identify this token later.
terraform login

Expiration: 30 days
This token will expire June 23rd, 2023

Figure 14.2: Create TFC user token

Update the **Description** value if you want (I left the default description), and click the **Generate token** button.

4. The Terraform Cloud page displays the generated token value, so copy the value of the generated token.
5. The following image shows the token value:



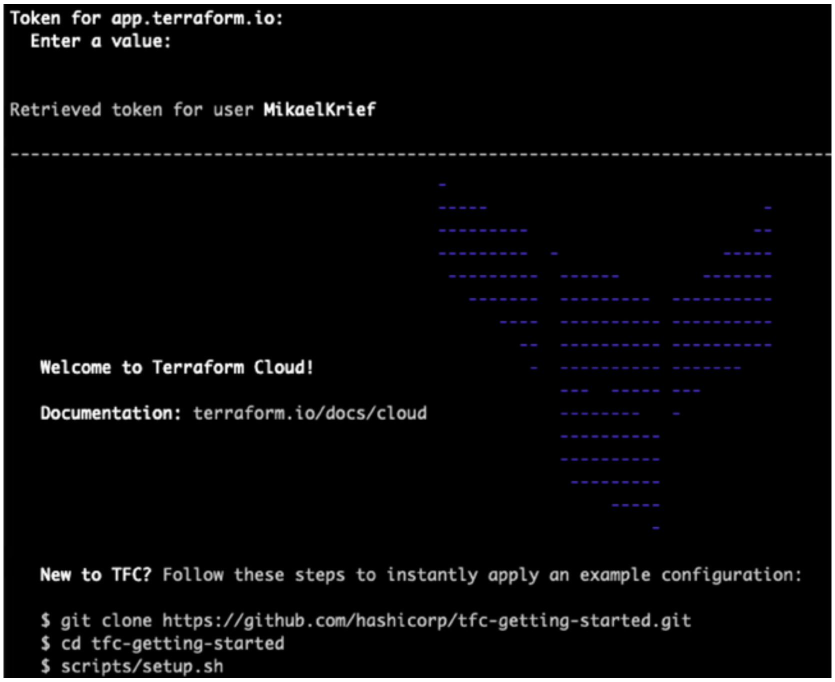
Tokens (4)

Click on the token to copy it, and paste into your Terraform login prompt to continue. This token **will not be displayed again**, so make sure to save it to a safe place.

```
2s7Bol9YIWzEbg.atlasv1.[redacted]
```

Figure 14.3: TFC token

6. Paste this token value in the terminal console opened in *Step 1* (the value will be hidden in the terminal).
7. Finally, at the end of the `terraform login` execution, Terraform displays a confirmation message about the successful authentication and the first steps to getting started with Terraform Cloud.



```
Token for app.terraform.io:
  Enter a value:


Retrieved token for user MikaelKrief

---------------------------------------------------------------------------------
                                          -
                                          -----                           -
                                          ---------                      --
                                          ---------  -                -----
                                           ---------  ------        -------
                                             -------  ---------  ----------
                                                ----  ----------  ----------
                                                  --  ----------  ----------
   Welcome to Terraform Cloud!                     -  ----------  -------
                                                      ---  -----  ---
   Documentation: terraform.io/docs/cloud             --------   -
                                                      ----------
                                                      ----------
                                                       ---------
                                                           -----
                                                               -


   New to TFC? Follow these steps to instantly apply an example configuration:

   $ git clone https://github.com/hashicorp/tfc-getting-started.git
   $ cd tfc-getting-started
   $ scripts/setup.sh
```

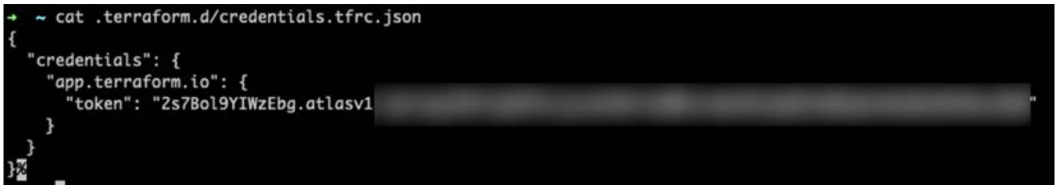
Figure 14.4: TFC authentication output

We are now authenticated to Terraform Cloud via Terraform CLI.

There's more...

In this recipe, we learned how to authenticate with Terraform Cloud using the interactive mode in the terminal console. This operation can be useful for the first authentication. If you want to just update the token value (for example, after its expiration), rerun the `login` command.

We can also perform this authentication without the CLI by editing the local file directly in `.terraform.d/credentials.tfrc.json`.



```
→ ~ cat .terraform.d/credentials.tfrc.json
{
  "credentials": {
    "app.terraform.io": {
      "token": "2s7Bol9YIWzEbg.atlasv1[redacted]"
    }
  }
}%
```

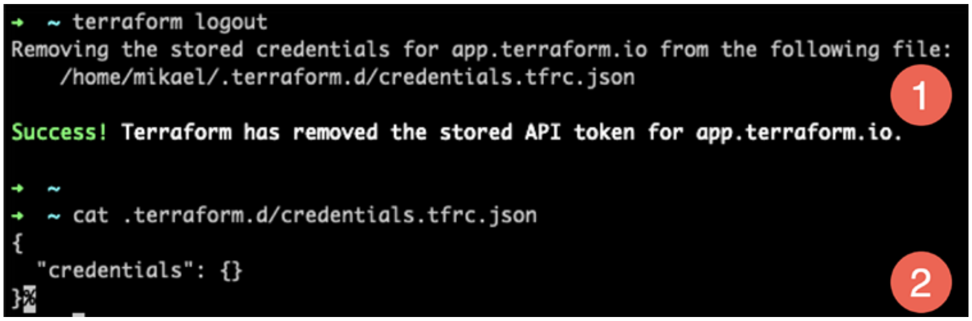
Figure 14.5: TFC credentials local file

We can see that the token value is stored in plain text in the `token` property.

Caution: This second method is less recommended as it can lead to manual errors in this JSON file.

Additionally, to log out, run the command `terraform logout` or delete the file at `.terraform.d/credentials.tfrc.json`.

The following image shows the two ways to log out of Terraform from Terraform Cloud:



```
→ ~ terraform logout
Removing the stored credentials for app.terraform.io from the following file:
    /home/mikael/.terraform.d/credentials.tfrc.json

Success! Terraform has removed the stored API token for app.terraform.io.

→ ~
→ ~ cat .terraform.d/credentials.tfrc.json
{
  "credentials": {}
}%
```

Figure 14.6: Terraform logout command

We can see in this image:

- In the area labeled 1, the output of the command `terraform logout`
- In the area labeled 2, the content of the file `.terraform.d/credentials.tfrc.json` for logging out

Another point is, if you want to use multiple credentials to authenticate with multiple hosts (the hostname of Terraform Cloud is `app.terraform.io`, but every Terraform Enterprise instance can have different hostnames), you can add multiple credential entries in the `terraform.rc` file as is detailed in this documentation here: `https://developer.hashicorp.com/terraform/cli/config/config-file#credentials-1`.

Moreover, we can bypass or override the credentials stored in `terraform.rc` by using a specific `TF_TOKEN_app_terraform_io` environment variable.

For more information about the Terraform Cloud environment variable, read the documentation here: `https://developer.hashicorp.com/terraform/cli/config/config-file#environment-variable-credentials`.

Additionally, if you want to list and manage all created user tokens, go to the page `https://app.terraform.io/app/settings/tokens`.

See also

- Documentation of the `terraform login` command can be found here: `https://developer.hashicorp.com/terraform/cli/commands/login`.
- Documentation of the `terraform` logout command can be found here: `https://developer.hashicorp.com/terraform/cli/commands/logout`.
- A tutorial on the `terraform login` command can be found here: `https://developer.hashicorp.com/terraform/tutorials/cloud-get-started/cloud-login`.

Managing workspaces in Terraform Cloud

One of the main components of Terraform Cloud is a workspace. TFC workspaces enable users to organize the provisioning of infrastructure components inside logical groups or components. For example, we can decouple Terraform components to have one component for provisioning the network and a second component to provision an Azure VM by creating two TFC workspaces in Terraform Cloud.

In addition to workspaces, we can create TFC projects to organize multiple TFC workspaces inside.

Very important note: Be careful not to confuse the Terraform CLI workspaces we learned about in *Chapter 6*, *Applying a Basic Terraform Workflow*, in the recipe *Using workspaces for managing environments*, with the TFC workspaces that we cover in this recipe.

For more details on the differences, read the documentation here: `https://developer.hashicorp.com/terraform/cloud-docs/workspaces#terraform-cloud-vs-terraform-cli-workspaces`.

Let's get started!

Getting ready

To complete this recipe, you'll need to have already created a Terraform Cloud organization.

Additionally, before creating workspaces, it's important to check the required permissions. For this, read this documentation: `https://developer.hashicorp.com/terraform/cloud-docs/workspaces/creating#permissions`.

The goal of this recipe is to create one project and two workspaces within it, using the Terraform Cloud UI.

How to do it…

To create a TFC workspace, perform the following steps:

1. The first step is to create the TFC project. For this, click the link **Projects & workspaces** in the left menu, then click on the **New** button, and then on the **Project** button.

   The following image shows the steps to create a project:

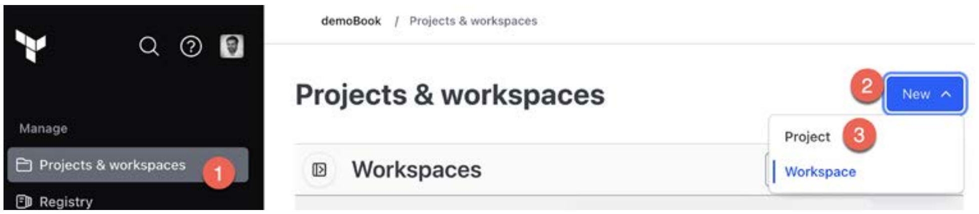


Figure 14.7: TFC create project

2. In the project form, type a project name, such as **DemoVM**, and click on the **Create** button as shown in the following image:

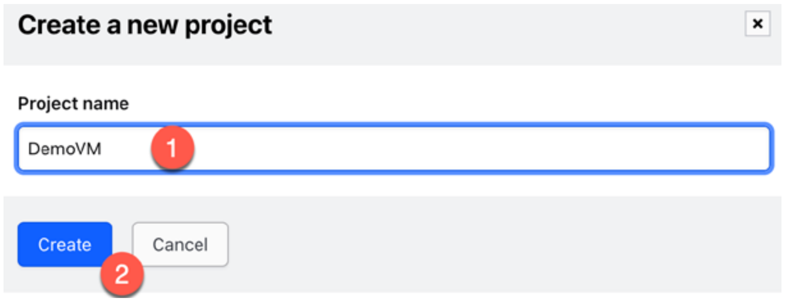


Figure 14.8: TFC set project name

3. Then, in the second part, we create the first workspace inside this project by clicking the **New** > **Workspace** button:

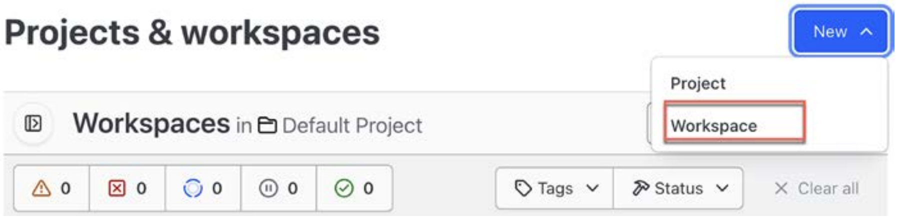
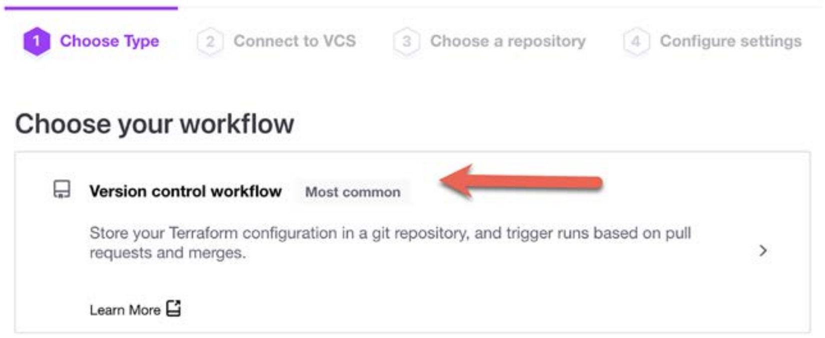


Figure 14.9: TFC create workspace

4. The workspace creation wizard takes you through the following steps:

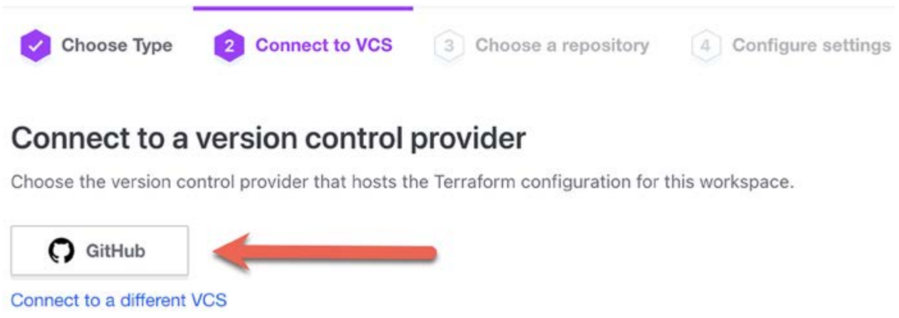
   a. The first step of the workspace wizard is to choose the workflow of the workspace; we will choose the first option to use the version control workflow:



Store your Terraform configuration in a git repository, and trigger runs based on pull requests and merges.

Figure 14.10: TFC workspace version control

b. Then, we choose **GitHub** as the source control:

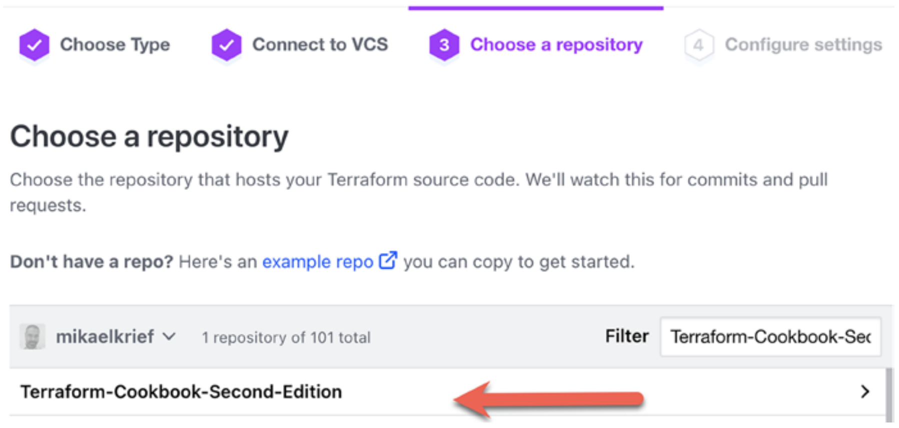


Connect to a version control provider

Choose the version control provider that hosts the Terraform configuration for this workspace.

Figure 14.11: Terraform Cloud connecting workspace to GitHub

c. Then, we select the GitHub repository that contains the Terraform configuration (here we've selected the fork of the repository of this book).



Choose a repository

Choose the repository that hosts your Terraform source code. We'll watch this for commits and pull requests.

Don't have a repo? Here's an example repo you can copy to get started.

1 repository of 101 total

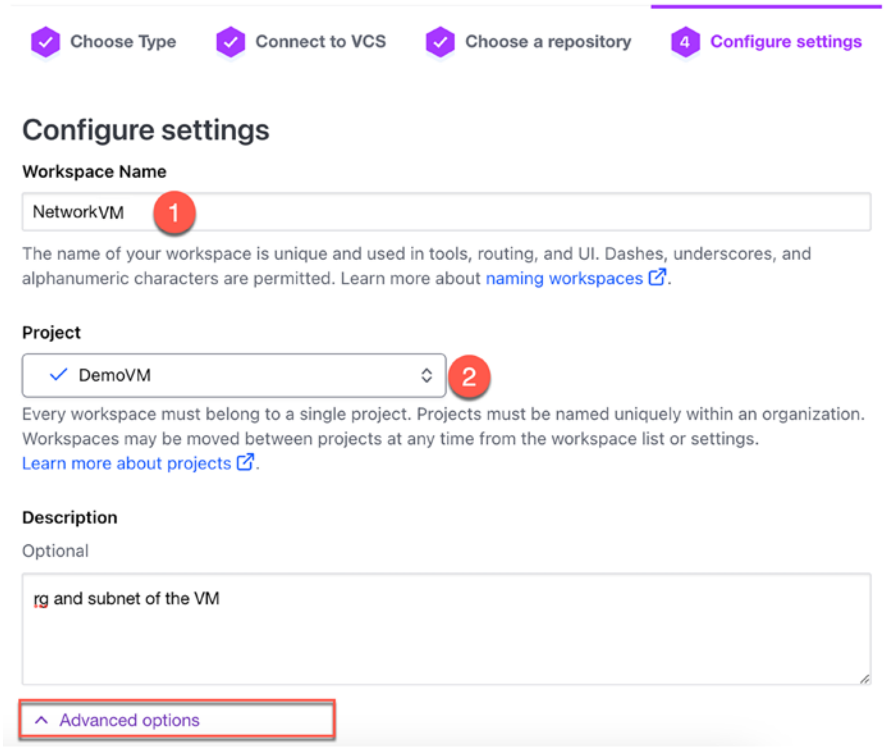
Filter: Terraform-Cookbook-Sec

Terraform-Cookbook-Second-Edition

Figure 14.12: Selection of a GitHub repository

Finally, we need to fill out the settings of the workspace.

d. Input the name of the workspace (read the workspace name nomenclature constraints here: `https://developer.hashicorp.com/terraform/cloud-docs/workspaces/creating#workspace-naming`), and select in which project it will be included. Here, we chose the project **DemoVM**, created in *Steps 1* and *2* of this recipe.



Configure settings

Workspace Name

NetworkVM

The name of your workspace is unique and used in tools, routing, and UI. Dashes, underscores, and alphanumeric characters are permitted. Learn more about naming workspaces.

Project

✓ DemoVM

Every workspace must belong to a single project. Projects must be named uniquely within an organization. Workspaces may be moved between projects at any time from the workspace list or settings. Learn more about projects.

Description

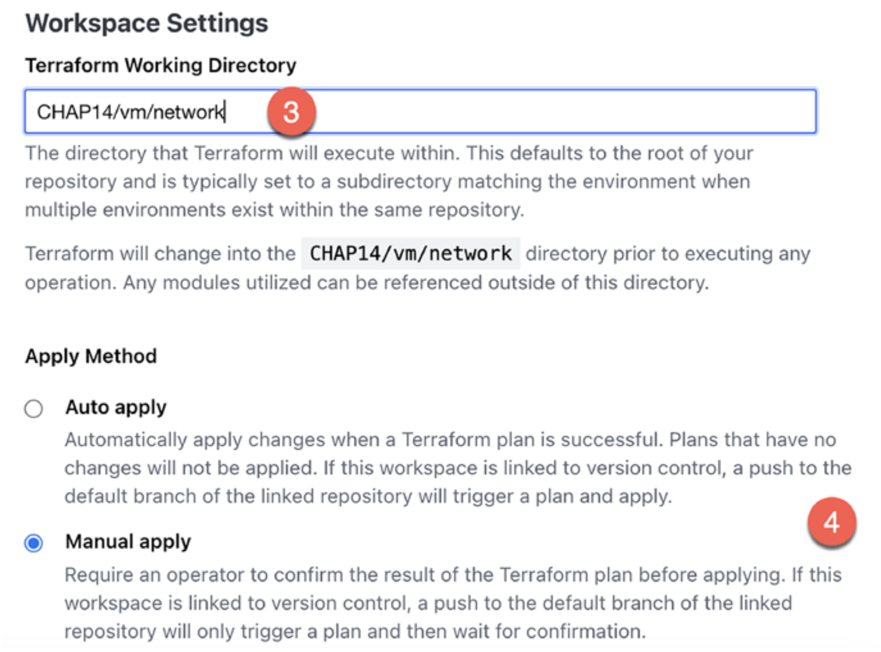
Optional

rg and subnet of the VM

Figure 14.13: Workspace name configuration

e. Click the **Advanced options** link for more settings.

f. Input the path to the Terraform configuration inside the repository, and choose **Manual apply** to check the plan before applying it:


Workspace Settings

Terraform Working Directory

CHAP14/vm/network

The directory that Terraform will execute within. This defaults to the root of your repository and is typically set to a subdirectory matching the environment when multiple environments exist within the same repository.

Terraform will change into the `CHAP14/vm/network` directory prior to executing any operation. Any modules utilized can be referenced outside of this directory.

Apply Method

○ Auto apply

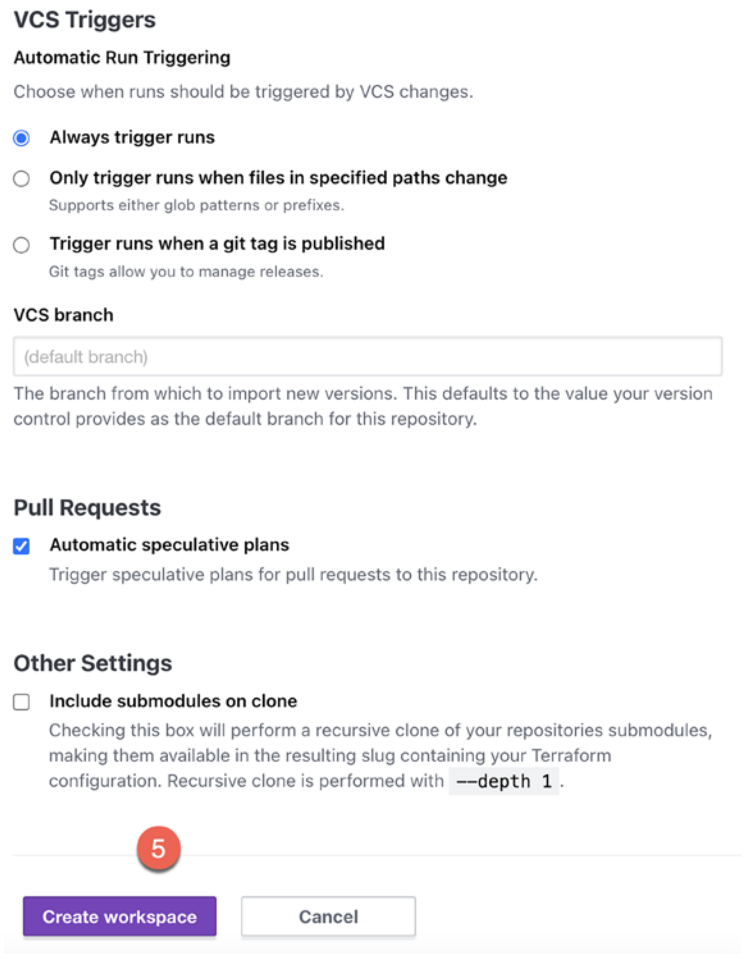
Automatically apply changes when a Terraform plan is successful. Plans that have no changes will not be applied. If this workspace is linked to version control, a push to the default branch of the linked repository will trigger a plan and apply.

◉ Manual apply

Require an operator to confirm the result of the Terraform plan before applying. If this workspace is linked to version control, a push to the default branch of the linked repository will only trigger a plan and then wait for confirmation.

Figure 14.14: Workspace Terraform configuration

g. Leave the other default configurations and click the **Create workspace** button.



VCS Triggers

Automatic Run Triggering

Choose when runs should be triggered by VCS changes.

● Always trigger runs

○ Only trigger runs when files in specified paths change
Supports either glob patterns or prefixes.

○ Trigger runs when a git tag is published
Git tags allow you to manage releases.

VCS branch

(default branch)

The branch from which to import new versions. This defaults to the value your version control provides as the default branch for this repository.

Pull Requests

✓ Automatic speculative plans
Trigger speculative plans for pull requests to this repository.

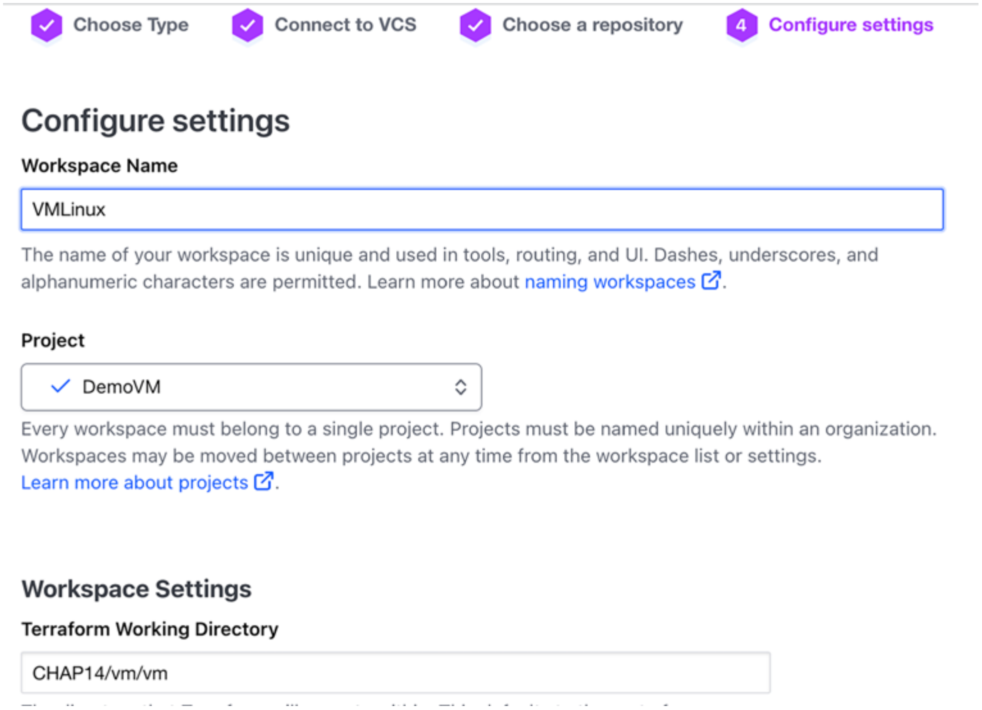
Other Settings

☐ Include submodules on clone
Checking this box will perform a recursive clone of your repositories submodules, making them available in the resulting slug containing your Terraform configuration. Recursive clone is performed with `--depth 1`.

Figure 14.15: Triggers configuration

The first workspace is now created.

5. To create the second workspace, perform the same operation as in *Steps 3* and *4* with the workspace name **VMLinux** and the path to the Terraform configuration of the VM.



Configure settings

Workspace Name

VMLinux

The name of your workspace is unique and used in tools, routing, and UI. Dashes, underscores, and alphanumeric characters are permitted. Learn more about naming workspaces.

Project

DemoVM

Every workspace must belong to a single project. Projects must be named uniquely within an organization. Workspaces may be moved between projects at any time from the workspace list or settings. Learn more about projects.

Workspace Settings

Terraform Working Directory

CHAP14/vm/vm

Figure 14.16: TFC second workspace configuration

6. We can see the list and status of the created project and associated workspaces, as shown in the following image, which includes the steps to display a workspace's list of projects:

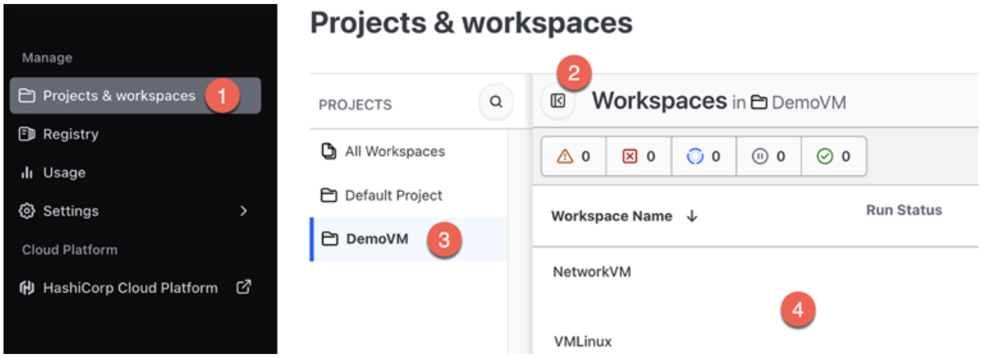
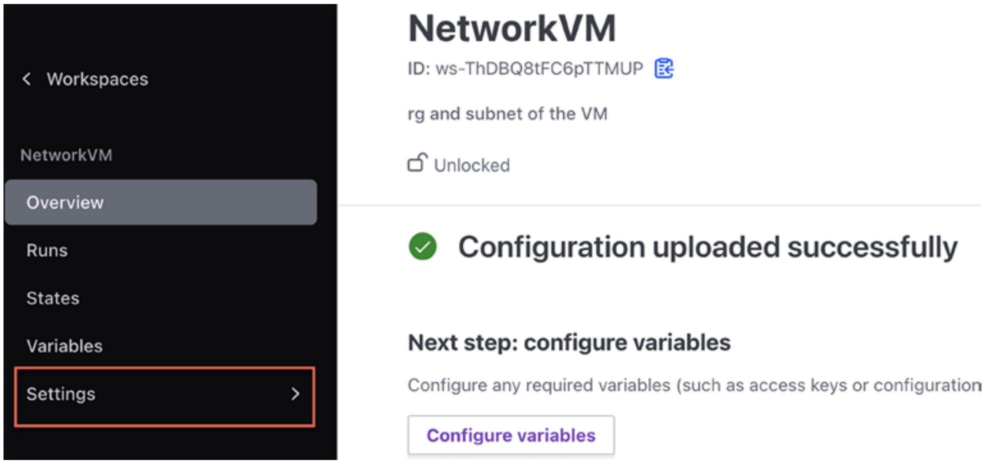


Figure 14.17: TFC workspace list in project

7. To manage the settings of specific TFC workspaces, click on the desired workspace and click the **Settings** button in the left menu:



NetworkVM

ID: ws-ThDBQ8tFC6pTTMUP

rg and subnet of the VM

Unlocked

Configuration uploaded successfully

Next step: configure variables

Configure any required variables (such as access keys or configuration

Figure 14.18: TFC workspace Settings menu option

In these settings, we can update all the workspace configurations and also delete the workspace by clicking on the **Destruction and deletion** left menu option.

How it works…

At the end of the execution of this recipe, we get the following project/workspace structure:

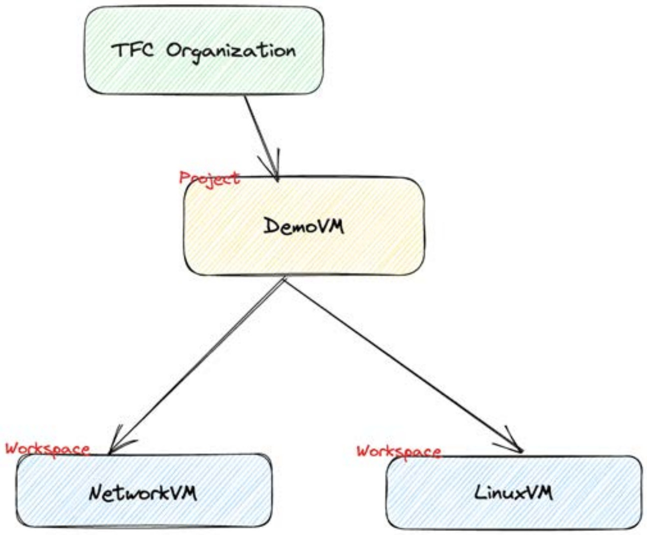


Figure 14.19: TFC workspace organization

In the next recipes of this chapter, we will learn in detail how to use these created workspaces.

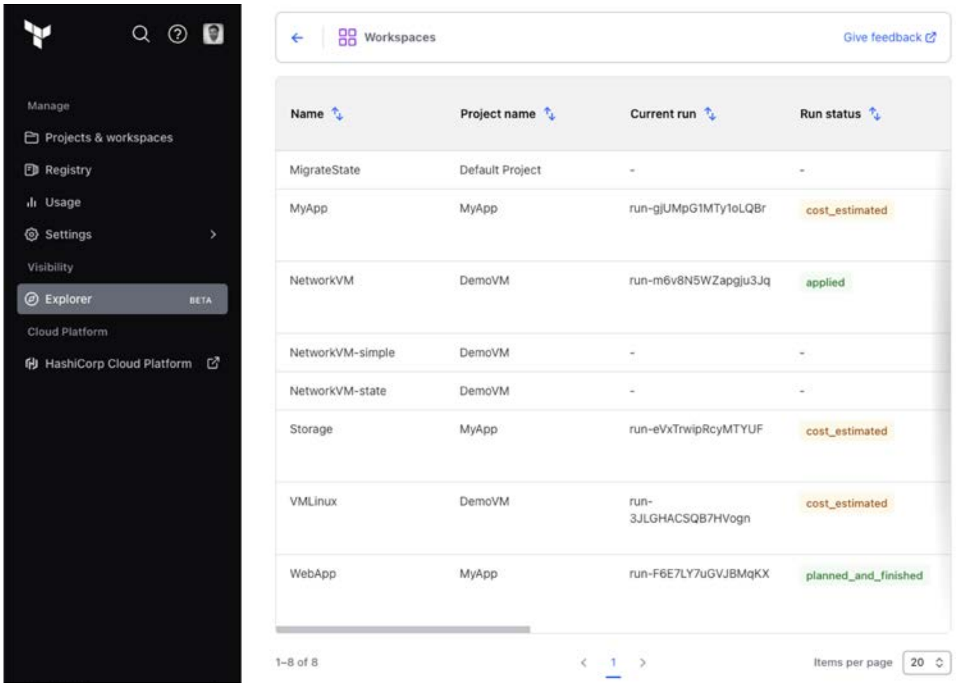
There's more...

In this recipe, we learned how to create a TFC workspace using the TFC UI, which provides friendly and intuitive UI steps.

In an automation context, we can also create a TFC workspace using either the TFC API (for more information, refer to the API documentation here: `https://developer.hashicorp.com/terraform/cloud-docs/api-docs/workspaces#create-a-workspace`) or the Terraform configuration and the Terraform Enterprise/Cloud provider (called `tfe`), which we will learn about in more detail in the recipe *Configuring Terraform Cloud with the Terraform TFE provider* later in this chapter.

Since it was announced in HashiDays 2023, Terraform Cloud has also included an organization explorer (in beta at the time this chapter is being written). The organization explorer enables you to see a consolidated view of the components in your organization, such as the workspace dashboard and the Terraform version that is being used by your Terraform configuration.

An excerpt from the workspace dashboard is shown in the image below:

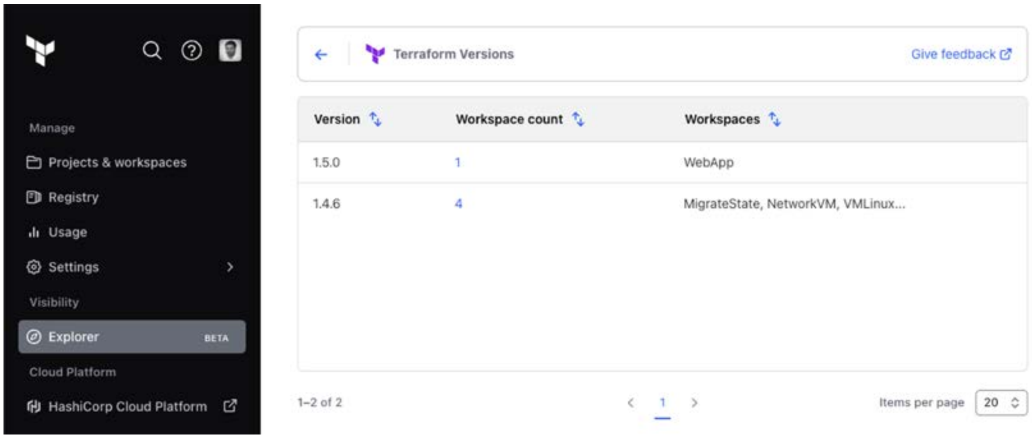


Name	Project name	Current run	Run status
MigrateState	Default Project	-	-
MyApp	MyApp	run-gjUMpG1MTy1oLQBr	cost_estimated
NetworkVM	DemoVM	run-m6v8N5WZapgju3Jq	applied
NetworkVM-simple	DemoVM	-	-
NetworkVM-state	DemoVM	-	-
Storage	MyApp	run-eVxTrwipRcyMTYUF	cost_estimated
VMLinux	DemoVM	run-3JLGHACSQB7HVogn	cost_estimated
WebApp	MyApp	run-F6E7LY7uGVJBMqKX	planned_and_finished

1–8 of 8 1 Items per page 20

Figure 14.20: TFC explorer workspace list

And the following image shows the Terraform provider dashboard:



Version	Workspace count	Workspaces
1.5.0	1	WebApp
1.4.6	4	MigrateState, NetworkVM, VMLinux...

Figure 14.21: TFC explorer Terraform version list

For more details about the Explorer view feature, read the documentation here, `https://developer.hashicorp.com/terraform/cloud-docs/workspaces/explorer`, and the blog post at `https://www.hashicorp.com/blog/new-terraform-cloud-capabilities-to-import-view-and-manage-infrastructure`.

See also

- Documentation on TFC projects and workspaces is available here: `https://developer.hashicorp.com/terraform/cloud-docs/workspaces` and `https://developer.hashicorp.com/terraform/cloud-docs/workspaces/organize-workspaces-with-projects`.
- A tutorial about creating TFC workspaces is available here: `https://developer.hashicorp.com/terraform/cloud-docs/workspaces/creating`.
- A video from Ned Bellavance about Terraform Cloud workspaces can be found here: `https://www.youtube.com/watch?v=tDexI54Cjs8`.

Using the remote backend in Terraform Cloud

Throughout this book, we have discussed storing and sharing the Terraform state.

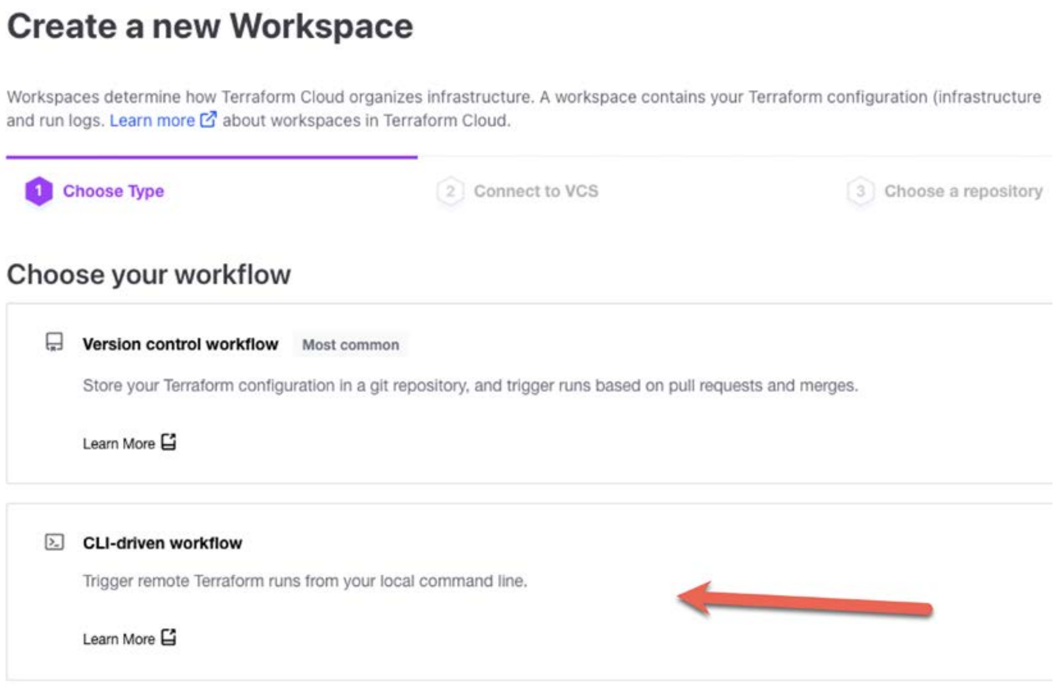
In the *Protecting the state file in the Azure remote backend* recipe in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform,* we saw a concrete case of this when we set up and used a backend in Azure (using Azure Storage). However, this recipe can only be applied with an Azure subscription. The different types of backends listed at `https://www.terraform.io/docs/backends/types/index.html` mostly require you to purchase platforms or tools.

One of Terraform Cloud's primary features is that it allows you to host the state.

In this recipe, we will learn how to use the `cloud` backend, which refers to Terraform Cloud.

Getting ready

The prerequisite for this recipe (as for all the others in this chapter) is that you have an account on Terraform Cloud (`http://app.terraform.io/`) and are logged in. Furthermore, you will need to create a project called `demoVM`, and inside of this project, create a workspace called `networkVM-state` by following the steps found in the *Managing workspaces in Terraform Cloud* recipe in this chapter, but selecting **CLI-driven workflow** for the workspace type.



Create a new Workspace

Workspaces determine how Terraform Cloud organizes infrastructure. A workspace contains your Terraform configuration (infrastructure and run logs. Learn more about workspaces in Terraform Cloud.

1 Choose Type — 2 Connect to VCS — 3 Choose a repository

Choose your workflow

Version control workflow Most common

Store your Terraform configuration in a git repository, and trigger runs based on pull requests and merges.

Learn More

CLI-driven workflow

Trigger remote Terraform runs from your local command line.

Learn More

Figure 14.22: TFC workspace CLI-driven workflow

The goal of this recipe is to configure and use the `cloud` backend for a simple Terraform configuration (that does not depend on a cloud infrastructure provider such as Azure). Furthermore, the execution of this configuration will be done in Terraform Cloud in **local mode**, that is, on a machine outside Terraform Cloud (which can be a local workstation or a CI/CD pipeline agent).

The source code for this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP14/vm/network`.

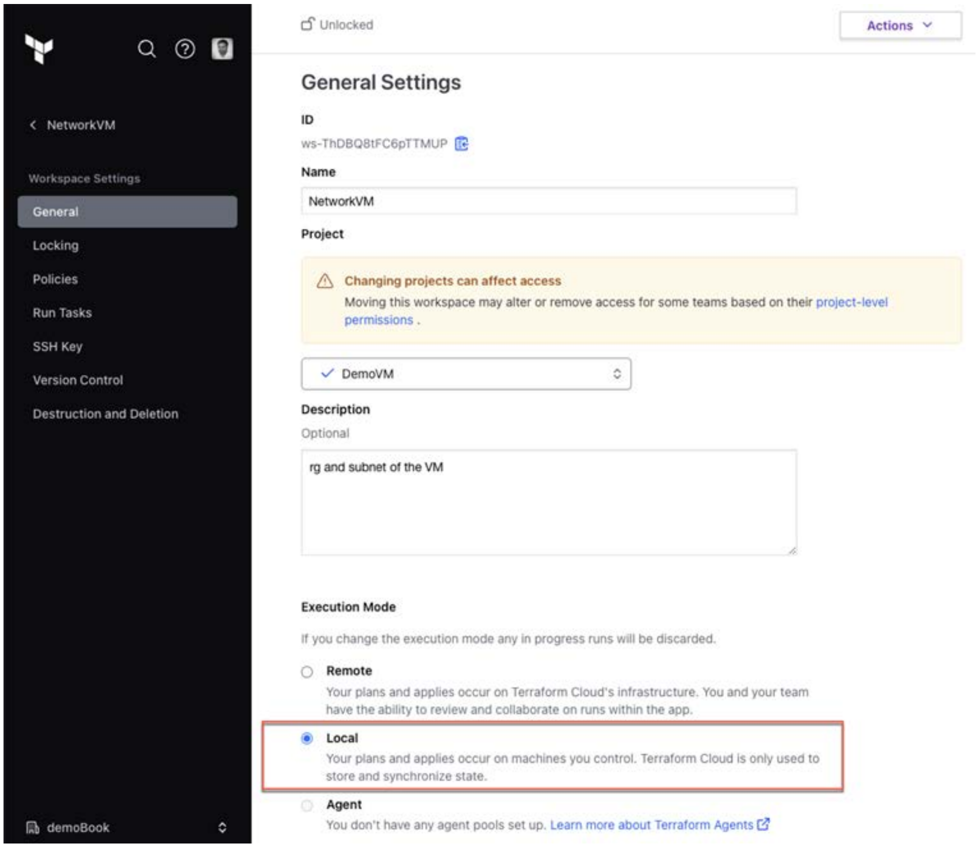
How to do it...

This recipe will be split into three parts, as follows:

1. Configuring local mode execution in Terraform Cloud
2. Generating a new user API token
3. Configuring and using the `cloud` backend

For the first part, we will configure local mode execution, as follows:

1. In our new Terraform Cloud workspace `networkVM`, go to **Settings** in the left menu, and then to the **General** tab, and change the **Execution Mode** option to **Local**:



General Settings

ID
ws-ThDBQ8tFC6pTTMUP

Name
NetworkVM

Project

Changing projects can affect access
Moving this workspace may alter or remove access for some teams based on their project-level permissions .

DemoVM

Description
Optional
rg and subnet of the VM

Execution Mode
If you change the execution mode any in progress runs will be discarded.

Remote
Your plans and applies occur on Terraform Cloud's infrastructure. You and your team have the ability to review and collaborate on runs within the app.

[✓ Local]
Your plans and applies occur on machines you control. Terraform Cloud is only used to store and synchronize state.

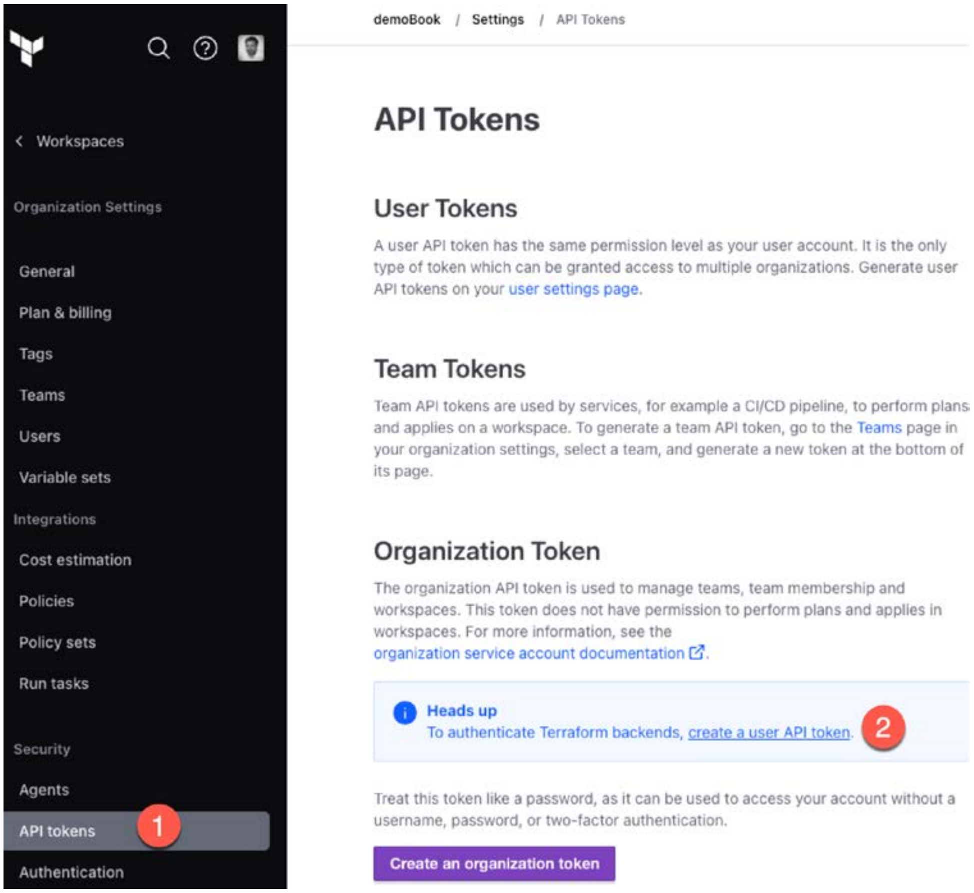
Agent
You don't have any agent pools set up. Learn more about Terraform Agents

Figure 14.23: TFC workspace local mode execution

2. Click the **Save settings** button to apply these changes.

Now, for the second part, we need to generate a new API token to authenticate with Terraform Cloud. Follow these steps:

1. In the **Settings** tab of the `demoBook` organization, go to the **API tokens** tab.
2. Scroll down to the bottom of this page and click on the **create a user API token** link to generate a new user API token.



demoBook / Settings / API Tokens

API Tokens

User Tokens

A user API token has the same permission level as your user account. It is the only type of token which can be granted access to multiple organizations. Generate user API tokens on your user settings page.

Team Tokens

Team API tokens are used by services, for example a CI/CD pipeline, to perform plans and applies on a workspace. To generate a team API token, go to the Teams page in your organization settings, select a team, and generate a new token at the bottom of its page.

Organization Token

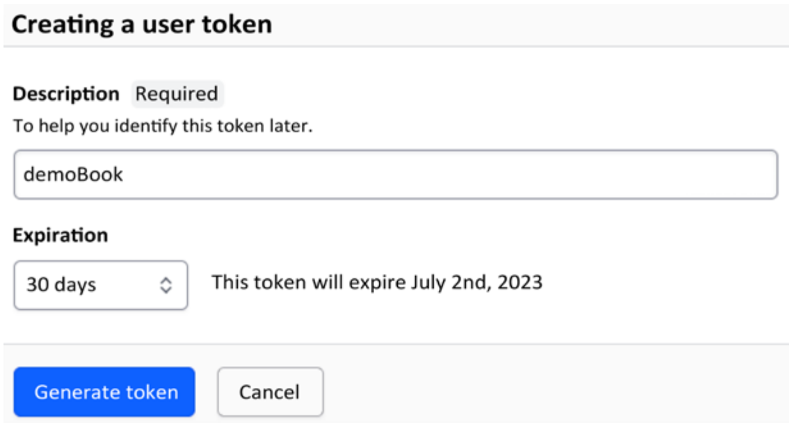
The organization API token is used to manage teams, team membership and workspaces. This token does not have permission to perform plans and applies in workspaces. For more information, see the organization service account documentation.

Heads up
To authenticate Terraform backends, create a user API token.

Treat this token like a password, as it can be used to access your account without a username, password, or two-factor authentication.

Figure 14.24: TFC API tokens

3. On the opened page, click on **Create an API token**. In the newly opened dialog, enter the name of the token and choose the time expiration of the token (the default is 30 days). Finally, click on the **Generate token** button.



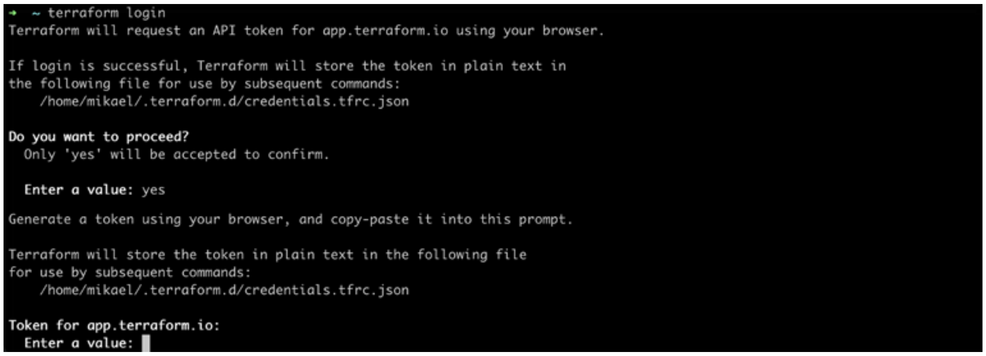
Creating a user token

Description Required
To help you identify this token later.
demoBook

Expiration
30 days — This token will expire July 2nd, 2023

Figure 14.25: TFC create a user token

4. The token is displayed in the token list on the page. Make a note of it as it will not be displayed after this page is reloaded.
5. Then run the command `terraform login` and paste the token during the execution of this command:



```
➜ ~ terraform login
Terraform will request an API token for app.terraform.io using your browser.

If login is successful, Terraform will store the token in plain text in
the following file for use by subsequent commands:
    /home/mikael/.terraform.d/credentials.tfrc.json

Do you want to proceed?
  Only 'yes' will be accepted to confirm.

  Enter a value: yes

Generate a token using your browser, and copy-paste it into this prompt.

Terraform will store the token in plain text in the following file
for use by subsequent commands:
    /home/mikael/.terraform.d/credentials.tfrc.json

Token for app.terraform.io:
  Enter a value: 
```

Figure 14.26: Terraform login command asking for the token

Finally, the last part is to configure and use the `remote` backend. Follow these steps:

1. In the `main.tf` file of the Terraform configuration, add the following `cloud` block configuration:

```
terraform {
  cloud {
    hostname     = "app.terraform.io"
    organization = "demoBook"

    workspaces {
      name = "networkVM"
    }
  }
}
```

2. For cloud authentication, here using Azure, set the four Azure authentication environment variables covered in the *"Protecting the Azure credential provider"* recipe in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*.
3. Execute the basic Terraform workflow commands `init`, `plan`, and `apply` from your local workstation.

How it works...

In the first part of this recipe, we configured the execution mode of our workspace. In this configuration, we chose **local** mode, which indicates that Terraform will assume the configuration is available on a private machine (either a local development workstation or a CI/CD pipeline agent). In this case, the created workspace is just used to store the Terraform State.

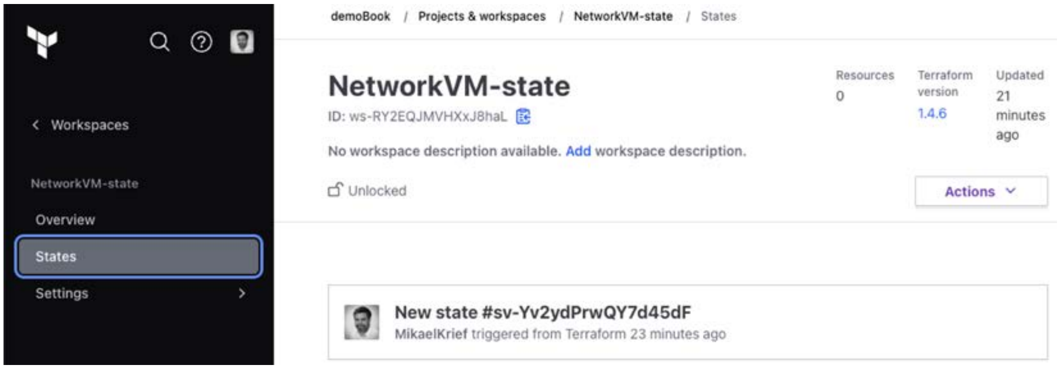
Then, in the second part, we created a token that allows the CLI to authenticate with our Terraform Cloud workspace.

In the last part, we wrote the configuration, which describes our TFC configuration that will store the state file. In this configuration, we used the `cloud` block, in which we added the following parameters:

- `hostname` with the `"app.terraform.io"` value, which is the domain of Terraform Cloud
- `organization`, which contains the `demoBook` name of the organization
- `workspaces` with the name of the `NetworkVM` workspace that we created manually in the prerequisites of this recipe

At the end, we executed the Terraform workflow commands locally.

After executing these commands, upon clicking on the **States** menu option of our workspace, we will see that our status file has been created:



NetworkVM-state
ID: ws-RY2EQJMVHXxJ8haL

Resources 0
Terraform version 1.4.6
Updated 21 minutes ago

No workspace description available. Add workspace description.

Unlocked

New state #sv-Yv2ydPrwQY7d45dF
MikaelKrief triggered from Terraform 23 minutes ago

Figure 14.27: TFC workspace Terraform state

By clicking on this file, you can view its content or download it.

There's more...

In this recipe, we created the User API token via the TFC UI and put it inside the CLI configuration file `terraform.rc`, which is the best file to use for automation mode.

In manual mode, we can use the `terraform login` command. In the terminal, run the command `terraform login` to authenticate Terraform to our workspace as we learned in the *Authenticating Terraform to Terraform Cloud* recipe in this chapter. During the execution of this command, enter the generated token.

To learn more about the use of API tokens, please refer to the documentation at `https://www.terraform.io/docs/cloud/users-teams-organizations/api-tokens.html`.

In the next recipe, we will learn how to migrate your existing state to Terraform Cloud.

See also

- The documentation on the `cloud` block is available here: `https://developer.hashicorp.com/terraform/cli/cloud/settings`.

Migrating Terraform State to Terraform Cloud

In the previous recipe, we learned how to use Terraform Cloud as a remote backend. In *Chapter 5*, *Managing Terraform State*, we learned how to operate the state. We will now combine both of these skills into one project. In this recipe, we will learn how to migrate the existing state to Terraform Cloud.

In general, when you migrate a state from one backend to another, if you don't follow the state migration procedure, Terraform will see a new backend in its configuration. Then, it will want to delete the resources, so you'll have to recreate them in the new state, which will cause a service interruption.

To avoid having to reprovision resources, we need to migrate the state to Terraform Cloud.

Let's get started!

Getting ready

To complete this recipe, you'll need to use the same version of Terraform CLI between the time of the original state creation and the time you will operate the migration of the state.

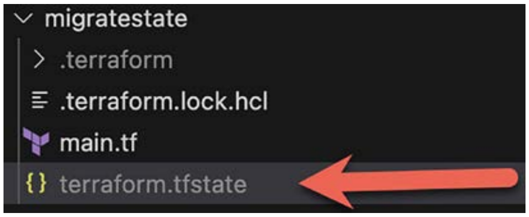
The migration of the state will be done with Terraform CLI and use the remote backend, so the Terraform version must be equal to or greater than 1.1.

Another requirement is to have a Terraform Cloud workspace already created that the new state will be imported to. For more details, read the first part of the previous recipe, *Using the remote backend in Terraform Cloud*.

In this recipe, we will use the Terraform configuration available here, `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP14/migratestate`, in which we have already performed the `terraform init`, `plan`, and `apply` commands.

> Note that the Terraform-created random password here is very basic, and is just to demonstrate the state migration.

After running the `apply` command, we can see the Terraform state file stored locally, as shown in the following image:



```
migratestate
  .terraform
  .terraform.lock.hcl
  main.tf
  terraform.tfstate
```

Figure 14.28: Terraform local state

The goal of this recipe is to migrate this existing state to Terraform Cloud without needing to destroy and redeploy the infrastructure.

How to do it...

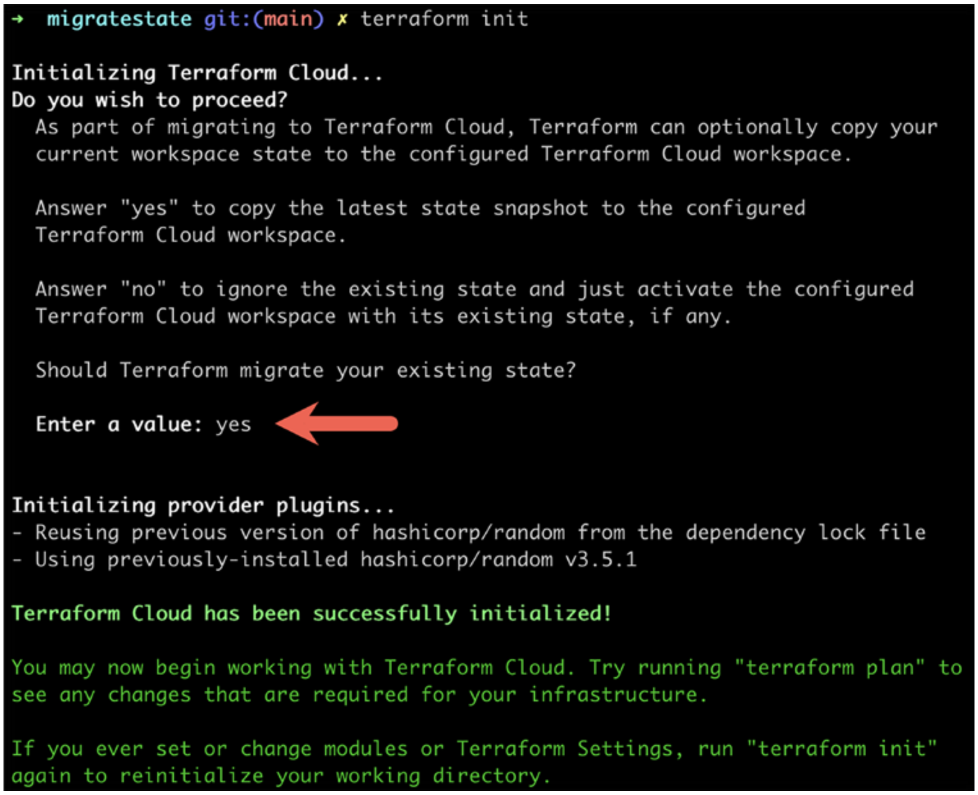
To migrate this existing state, perform the following steps:

1. In your existing configuration, configure the remote backend by adding the `cloud` backend configuration, as we learned in the previous recipe, by adding the following Terraform configuration in `main.tf`:

   ```
   terraform {
     required_version = "~> 1.1"
     required_providers {
   random = {
         source = "hashicorp/random"
         version = "3.5.1"
       }

     }
       cloud {
       hostname      = "app.terraform.io"
       organization = "demoBook"

       workspaces {
         name = "MigrateState"
       }
     }
   }
   ```

2. Inside the folder that contains this Terraform configuration, run the `terraform init` command. The `init` command will detect that the backend configuration has changed and ask you to confirm the migration of the state to Terraform Cloud.

3. We confirm the migration by entering yes. The following image shows the execution of the `terraform init` command:

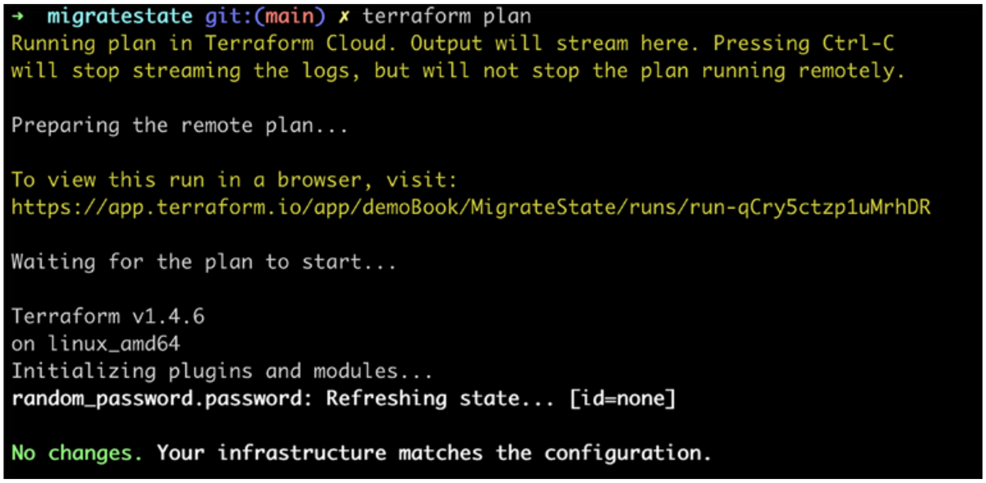


```
➜  migratestate git:(main) ✗ terraform init

Initializing Terraform Cloud...
Do you wish to proceed?
  As part of migrating to Terraform Cloud, Terraform can optionally copy your
  current workspace state to the configured Terraform Cloud workspace.

  Answer "yes" to copy the latest state snapshot to the configured
  Terraform Cloud workspace.

  Answer "no" to ignore the existing state and just activate the configured
  Terraform Cloud workspace with its existing state, if any.

  Should Terraform migrate your existing state?

  Enter a value: yes


Initializing provider plugins...
- Reusing previous version of hashicorp/random from the dependency lock file
- Using previously-installed hashicorp/random v3.5.1

Terraform Cloud has been successfully initialized!

You may now begin working with Terraform Cloud. Try running "terraform plan" to
see any changes that are required for your infrastructure.

If you ever set or change modules or Terraform Settings, run "terraform init"
again to reinitialize your working directory.
```

Figure 14.29: Terraform init migrate State

4. The migration of the state is done, so now we can delete the local `terraform.tfstate` created in the *Getting ready* section of this recipe.

5. Finally, we check the migration by running the `terraform plan` command in the same directory. The following image shows the output of the `terraform plan` command:



```
→  migratestate git:(main) ✗ terraform plan
Running plan in Terraform Cloud. Output will stream here. Pressing Ctrl-C
will stop streaming the logs, but will not stop the plan running remotely.

Preparing the remote plan...

To view this run in a browser, visit:
https://app.terraform.io/app/demoBook/MigrateState/runs/run-qCry5ctzp1uMrhDR

Waiting for the plan to start...

Terraform v1.4.6
on linux_amd64
Initializing plugins and modules...
random_password.password: Refreshing state... [id=none]

No changes. Your infrastructure matches the configuration.
```

Figure 14.30: Terraform plan after state migration

We can see that there are no changes, so the state is migrated without needing to destroy and recreate resources.

How it works...

In our recipe, the `terraform init` command detects the change of the Terraform configuration in the remote backend and then provides the choice to migrate the configuration into a Terraform Cloud workspace.

See also

- The State migration documentation is available here: `https://www.terraform.io/docs/cloud/migrate/index.html`.

Using Terraform Cloud as a private module registry

In the *Using the remote backend in Terraform Cloud* recipe *found in this chapter*, we learned how to use Terraform Cloud as a remote backend. It is centralized, secure, and free of charge.

In this book, we dedicated *Chapter 7, Sharing Terraform Configuration with Modules,* to the creation, usage, and sharing of Terraform modules. As a reminder, what we studied was publishing modules in the Terraform public registry, which is publicly accessible by all Terraform users, and sharing Terraform modules privately using a Git repository.

Concerning private module sharing, the Git repository system is efficient but does not offer a centralized platform for sharing and documenting modules as the public registry does. In order to provide companies with a private registry of their Terraform modules, HashiCorp has integrated this functionality into Terraform Cloud/Enterprise.

In this recipe, we will learn how to publish and use a Terraform module in the private registry of Terraform Cloud.

Getting ready

In order to publish a module in Terraform Registry, you'll need to store your module code in a VCS provider that is supported by Terraform Cloud. The list of supported file types can be found at `https://www.terraform.io/docs/cloud/vcs/index.html`.

To start this recipe, in the **Settings** section of the Terraform Cloud organization, we need to create a connection to the VCS provider that contains the Terraform configuration, as described in the documentation at `https://www.terraform.io/docs/cloud/vcs/index.html`.

In our scenario, we will use the GitHub VCS, which contains a `terraform-azurerm-webapp` repository (which creates an App Service plan, an App Service instance, and Application Insights in Azure).

To use this, you must fork this GitHub repository: `https://github.com/mikaelkrief/terraform-azurerm-webapp`.

(The Terraform configuration isn't updated with the latest version of the `azurerm` provider; it just serves as an example.)

In addition, as we studied in the *Sharing Terraform modules using GitHub* recipe in *Chapter 7, Sharing Terraform Configuration with Modules*, you need to create a Git tag in this repository that contains the version number of the module. For this recipe, we will create a `v1.0.0` tag, as shown in the following screenshot:

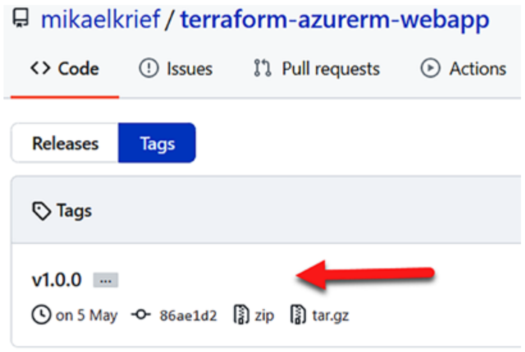
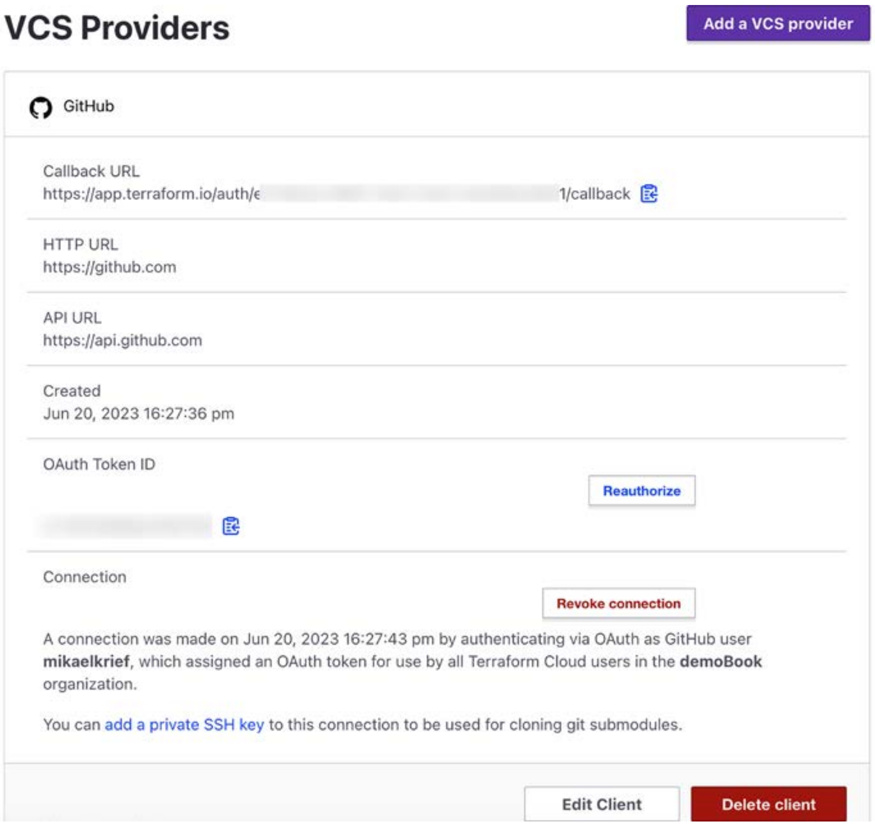


Figure 14.31: GitHub tag on the Terraform module

To integrate Terraform Cloud and GitHub, execute the process documented here: `https://www.terraform.io/docs/cloud/vcs/github-app.html`. At the end of this integration, we get the following screen when going to **Settings** > **VCS Providers**:


VCS Providers

Add a VCS provider

GitHub

Callback URL
https://app.terraform.io/auth/e[redacted]1/callback

HTTP URL
https://github.com

API URL
https://api.github.com

Created
Jun 20, 2023 16:27:36 pm

OAuth Token ID
[redacted]

Reauthorize

Connection

Revoke connection

A connection was made on Jun 20, 2023 16:27:43 pm by authenticating via OAuth as GitHub user **mikaelkrief**, which assigned an OAuth token for use by all Terraform Cloud users in the **demoBook** organization.

You can add a private SSH key to this connection to be used for cloning git submodules.

Edit Client

Delete client

Figure 14.32: TFC VCS provider

Our organization now has a connection to the required GitHub account and we can start publishing the module in Terraform Cloud.

How to do it…

To publish a Terraform module in Terraform Cloud's private registry, perform the following steps:

1. In our Terraform Cloud organization, click on the **Registry** menu, which is in the left bar menu, and click on the **Publish a module** link:

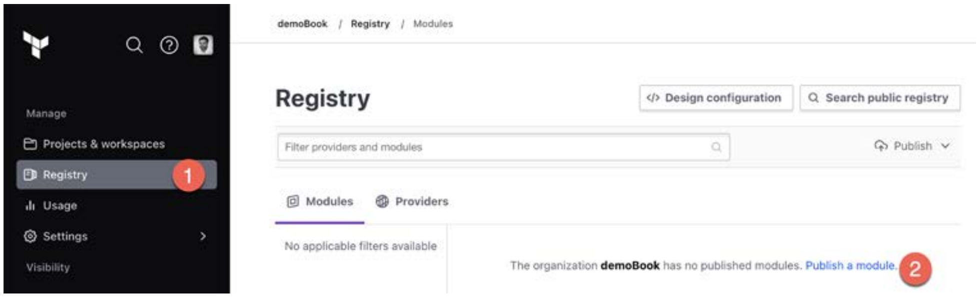
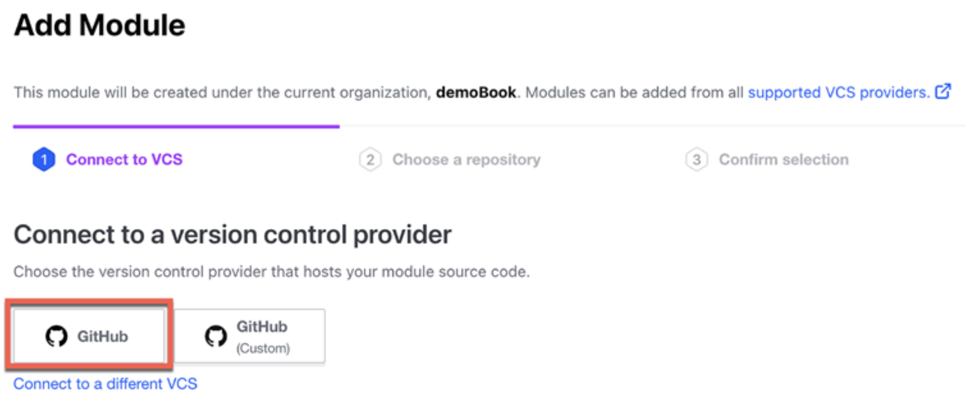

demoBook / Registry / Modules
Manage
Projects & workspaces
Registry
1
Usage
Settings
Visibility
Registry
Design configuration
Search public registry
Filter providers and modules
Publish
Modules
Providers
No applicable filters available
The organization demoBook has no published modules. Publish a module.
2

Figure 14.33: Publishing a module link in TFC

2. On the next page, in the first step of the wizard, choose **GitHub** as the VCS provider, which we integrated as part of the preparation for this recipe:

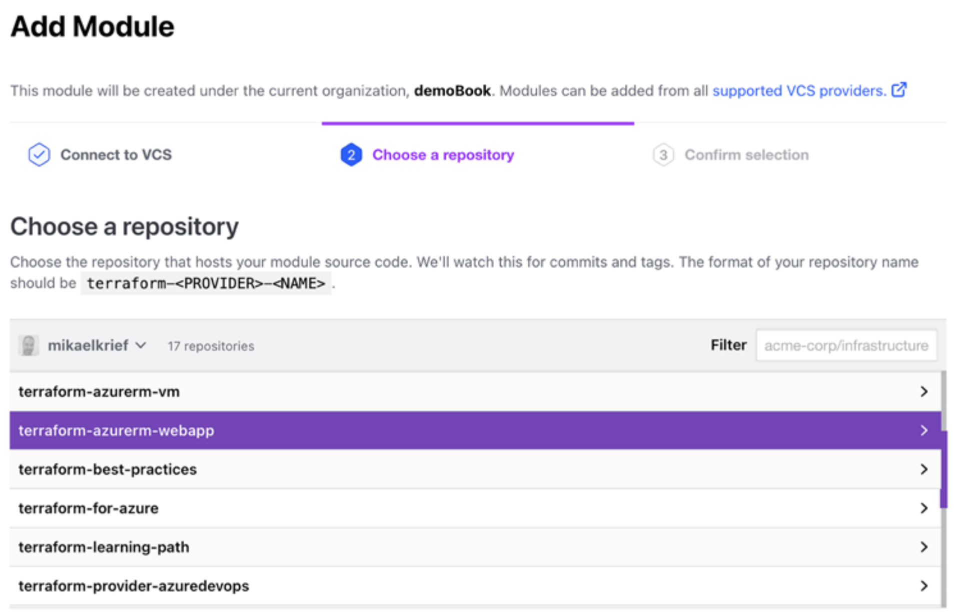


This module will be created under the current organization, **demoBook**. Modules can be added from all supported VCS providers.

Choose the version control provider that hosts your module source code.

Figure 14.34: Adding a module from GitHub in TFC

3. In the second step of the wizard, choose the repository that contains the Terraform module configuration:



Add Module

This module will be created under the current organization, **demoBook**. Modules can be added from all supported VCS providers.

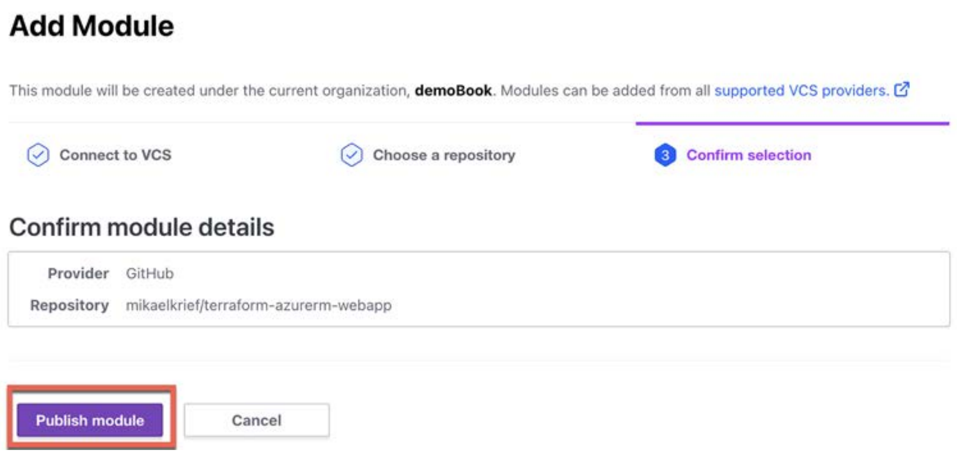
Choose a repository

Choose the repository that hosts your module source code. We'll watch this for commits and tags. The format of your repository name should be `terraform-<PROVIDER>-<NAME>`.

terraform-azurerm-vm
terraform-azurerm-webapp
terraform-best-practices
terraform-for-azure
terraform-learning-path
terraform-provider-azuredevops

Figure 14.35: Choosing a repository for your code in TFC

4. Finally, in the last step of the wizard, publish the module by clicking on the **Publish module** button:



Add Module

This module will be created under the current organization, **demoBook**. Modules can be added from all supported VCS providers.

Confirm module details

Provider	GitHub
Repository	mikaelkrief/terraform-azurerm-webapp

Figure 14.36: Publishing a Terraform module

The Terraform module is published on the private Terraform Cloud registry.

How it works...

To publish a module in the private registry of Terraform, you just have to follow the steps proposed by the wizard, which consists of choosing a VCS provider and then selecting the repository that contains the Terraform configuration of the module so that it can be published. After doing this, details about the module will be displayed in the layout of the public registry. In the center of this page, you will be able to see the contents of the `Readme.md` file, while on the right-hand side, you will be able to see the technical information about the use of this module.

There's more...

Once this module has been published in this registry, you can use it in a Terraform configuration. If you're using Terraform Cloud in local execution mode, you must configure Terraform CLI, as detailed in `https://developer.hashicorp.com/terraform/cli/config/config-file`. Then, you need to use this Terraform module in a Terraform configuration and write the following:

```
module "webapp" {
  source  = "app.terraform.io/<TFC organisation>/webapp/azurerm"
  version = "1.0.4"
...
}
```

In addition, if your modules have been published in this private registry, you can generate the Terraform configuration that calls them using the design configuration feature of Terraform Cloud. You can find out more about this at `https://www.terraform.io/docs/cloud/registry/design.html`.

Finally, please note that if you have several organizations in Terraform Cloud and you want to use the same private modules in all of them, you will have to publish these modules in each of your organizations. As for upgrading their versions, this will be done automatically for each organization.

See also

- Documentation regarding privately registering modules in Terraform Cloud is available here: `https://www.terraform.io/docs/cloud/registry/index.html`.

Executing Terraform configuration remotely in Terraform Cloud

In the *Managing workspaces in Terraform Cloud* recipe in this chapter, we studied how to create a workspace. During the Terraform Cloud workspace configuration, we can configure the CLI to apply the configuration present on a local workstation outside the Terraform Cloud platform. This private workstation is therefore private and can be a development workstation or a machine that serves as an agent for a CI/CD pipeline (such as on an Azure pipeline agent or a Jenkins node).

One of the great advantages of Terraform Cloud is its ability to execute Terraform configurations directly within this platform. This feature, called **remote operations**, makes it possible to have free Terraform configuration execution pipelines without having to install, configure, and maintain VMs that serve as agents. In addition, it provides a shared Terraform execution interface for all members of the organization.

In this recipe, we will look at the steps involved in running a Terraform configuration in Terraform Cloud using the UI workflow.

Getting ready

In this recipe, the Terraform configuration creates an Azure resource group and an Azure App Service instance.

Since, in this configuration, we will be creating Azure resources, we need to create an Azure service principal that has sufficient permissions in the subscription. For more information on Azure service principals and the authentication of Terraform in Azure, see the *Protecting the Azure credential provider* recipe in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform.*

Also, since we will be storing the configuration in GitHub, we will need to add the GitHub VCS provider, as explained in the documentation here: `https://www.terraform.io/docs/cloud/vcs/github-app.html`.

In your Terraform Cloud organization, you need to create a workspace (for this recipe, we will name this workspace `WebApp`). For more details on this procedure, read the *Managing workspaces in Terraform Cloud* recipe in this chapter.

Finally, all the steps of this recipe will be done in the Terraform Cloud web interface.

The Terraform configuration used in this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP14/remoteexec`.

Fork this repository to use this Terraform configuration in your Terraform Cloud organization and link your Terraform Cloud workspace with your forked repository using a VCS provider.

How to do it…

Because we're deploying resources in Azure, we first need to add the four Azure authentication environment variables to the workspace variable settings by performing the following steps:

1. On the `WebApp` workspace **Overview** page, click on the **Configure variables** button:

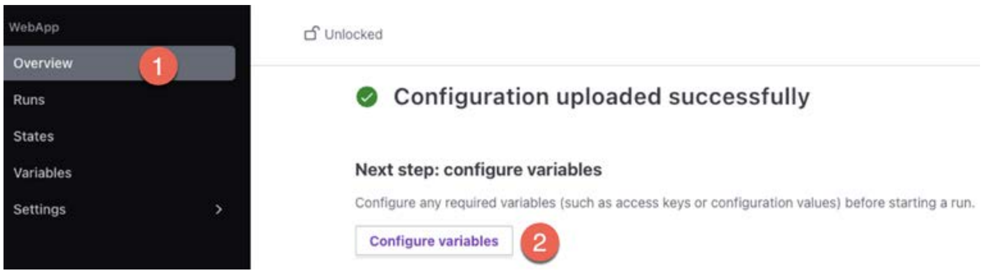

WebApp
Overview
1
Runs
States
Variables
Settings
Unlocked
Configuration uploaded successfully
Next step: configure variables
Configure any required variables (such as access keys or configuration values) before starting a run.
Configure variables
2

Figure 14.37: Configuring variables in the TFC workspace

2. Then, in the **Workspace variables** section, add our four Terraform `azurerm` provider authentication environment variables, as shown in the following screenshot:

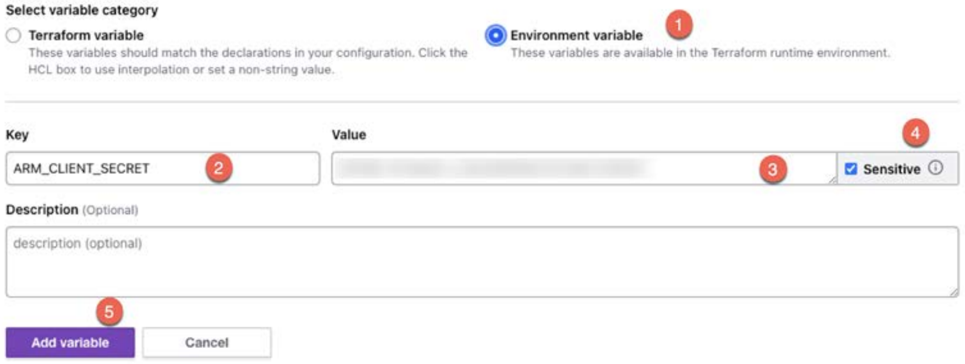
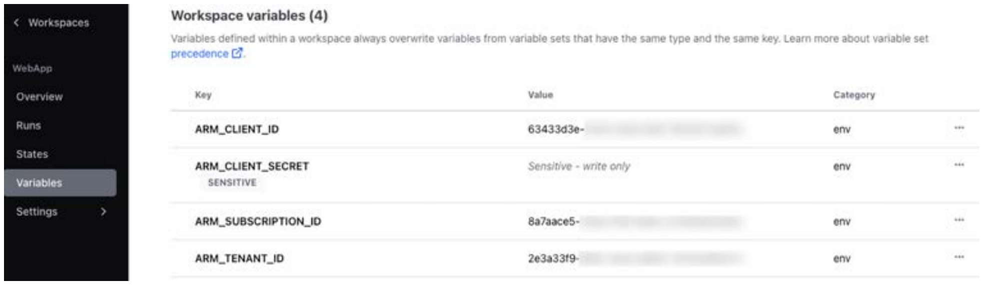

Select variable category
Terraform variable
These variables should match the declarations in your configuration. Click the HCL box to use interpolation or set a non-string value.
Environment variable
1
These variables are available in the Terraform runtime environment.
Key
ARM_CLIENT_SECRET
2
Value
3
4
Sensitive
Description (Optional)
description (optional)
5
Add variable
Cancel

Figure 14.38: Adding an environment variable in TFC

For each variable, configure the type of the variable (here, we set **Environment variable**), the variable name, and the value of the variable. Choose the **Sensitive** option for the secret and then click on the **Add variable** button.

3. Then, after creating the four environment variables, we can list them:



Workspace variables (4)

Variables defined within a workspace always overwrite variables from variable sets that have the same type and the same key. Learn more about variable set precedence.

Key	Value	Category
ARM_CLIENT_ID	63433d3e-	env
ARM_CLIENT_SECRET SENSITIVE	Sensitive - write only	env
ARM_SUBSCRIPTION_ID	8a7aace5-	env
ARM_TENANT_ID	2e3a33f9-	env

Figure 14.39: TFC variables list

Now that we've configured our workspace, we can execute the Terraform configuration inside Terraform Cloud.

To plan the configuration remotely in Terraform Cloud, perform the following steps:

1. To trigger the run, click on the **Actions** button and then on **Start new run**, as shown in the following image:

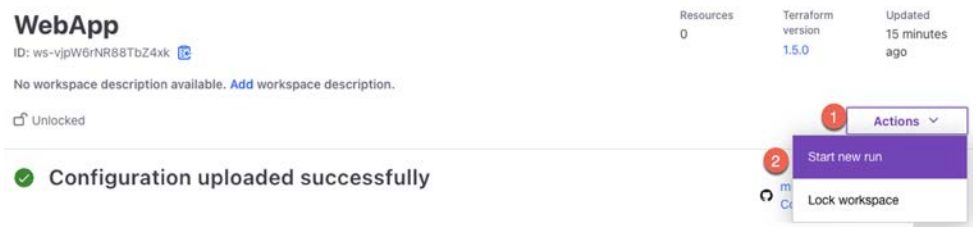
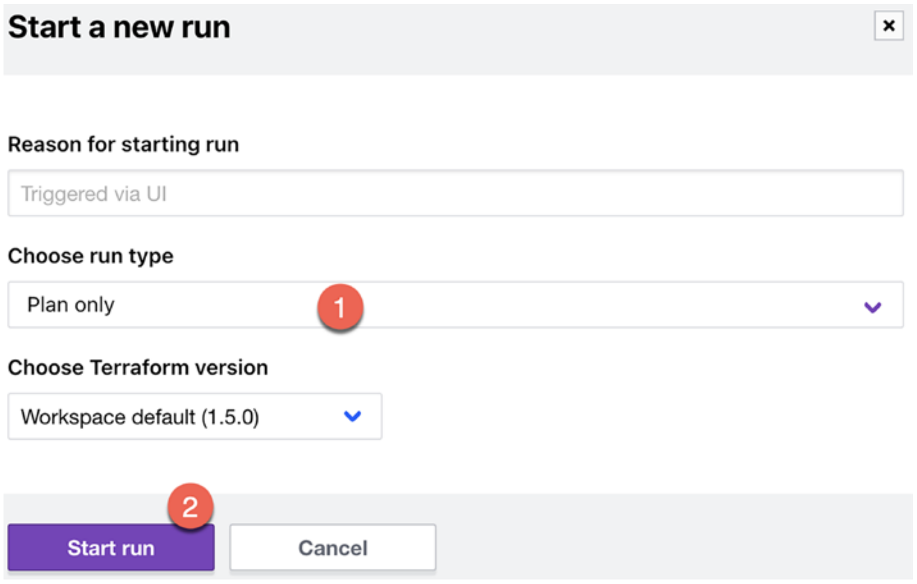


Figure 14.40: Start new run

2. In the window that opens, choose the option to run only the plan, and click on **Start run**.



Start a new run

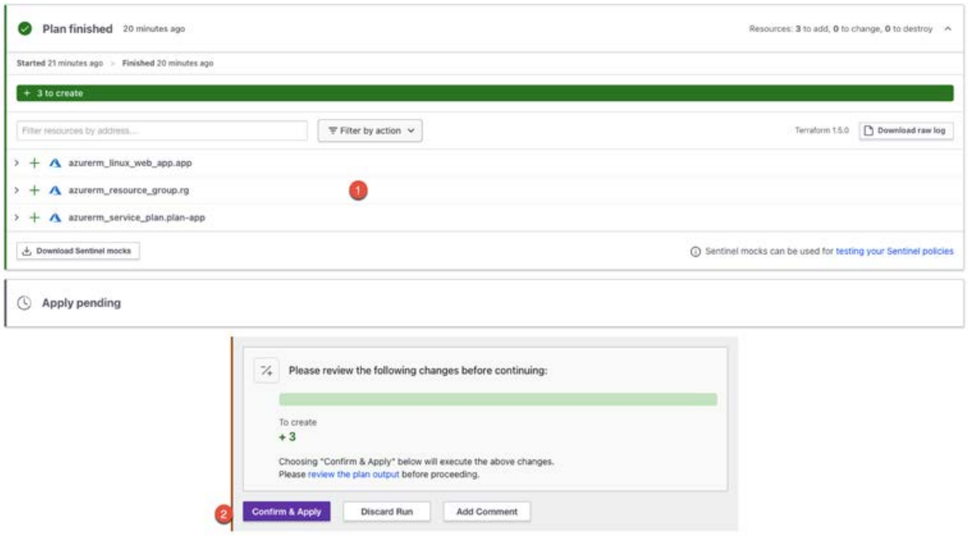
Reason for starting run: Triggered via UI

Choose run type: Plan only

Choose Terraform version: Workspace default (1.5.0)

Figure 14.41: TFC start run options

3. Terraform Cloud will launch a new execution for this Terraform configuration. By running the `terraform plan` command, we will be able to see the logs for this execution:



Plan finished 20 minutes ago

Resources: 3 to add, 0 to change, 0 to destroy

Started 21 minutes ago > Finished 20 minutes ago

+ 3 to create

Terraform 1.5.0

```
+ azurerm_linux_web_app.app
+ azurerm_resource_group.rg
+ azurerm_service_plan.plan-app
```

Sentinel mocks can be used for testing your Sentinel policies

Apply pending

Please review the following changes before continuing:

To create
+ 3

Choosing "Confirm & Apply" below will execute the above changes.
Please review the plan output before proceeding.

Figure 14.42: TFC run execution

4. After executing `plan`, Terraform Cloud expects the user to confirm this before the changes are applied.

   If we agree to the previewed changes, we can confirm these changes by clicking on the **Confirm & Apply** button:

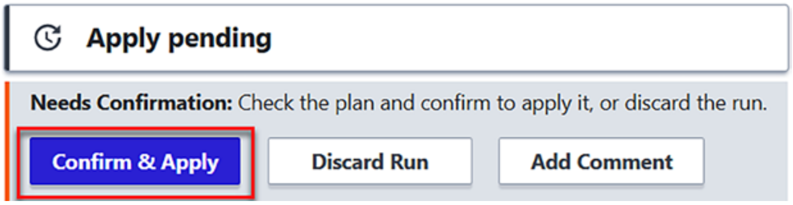

Apply pending
Needs Confirmation: Check the plan and confirm to apply it, or discard the run.
Confirm & Apply
Discard Run
Add Comment

Figure 14.43: Confirm and apply run

5. Add a comment and click on the **Confirm Plan** button:

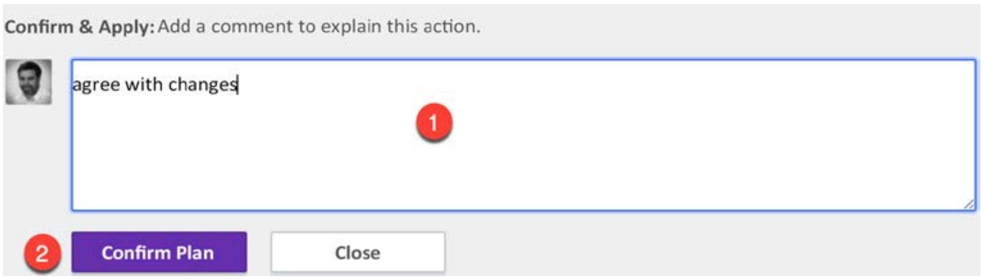
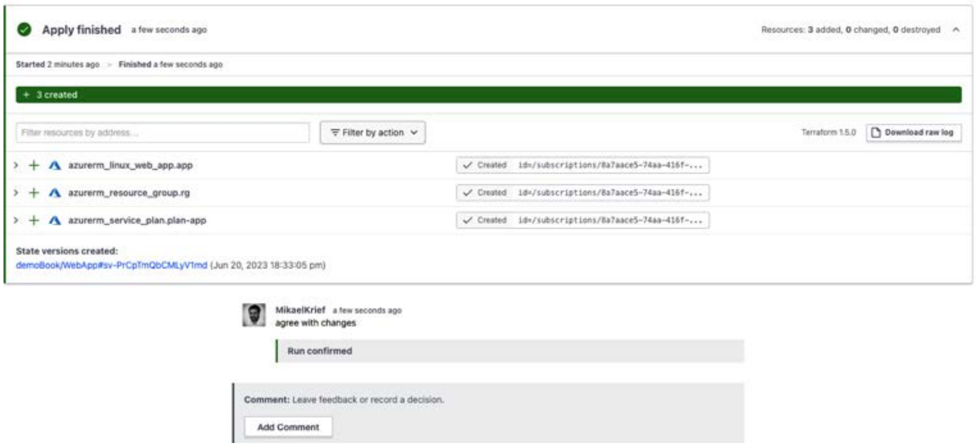


Figure 14.44: Confirm the plan to apply

6. Once finished, the result of executing the plan will be shown:



Apply finished a few seconds ago
Resources: 3 added, 0 changed, 0 destroyed

Started 2 minutes ago > Finished a few seconds ago

+ 3 created

Filter resources by address...
Filter by action
Terraform 1.5.0
Download raw log

azurerm_linux_web_app.app	✓ Created id=/subscriptions/8a7aace5-74aa-416f-...
azurerm_resource_group.rg	✓ Created id=/subscriptions/8a7aace5-74aa-416f-...
azurerm_service_plan.plan-app	✓ Created id=/subscriptions/8a7aace5-74aa-416f-...

State versions created:
demoBook/WebApp#sv-PrCpTmQbCMLyV1md (Jun 20, 2023 18:33:05 pm)

MikaelKrief a few seconds ago
agree with changes

Run confirmed

Comment: Leave feedback or record a decision.
Add Comment

Figure 14.45: TFC plan applied

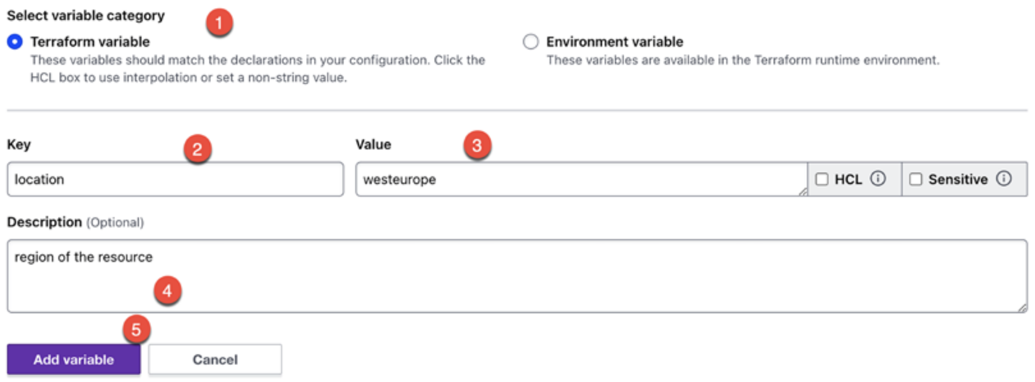
We can see the details of resource changes. Here three Azure resources will be created. At the end of this execution, our Azure resources are provisioned.

How it works...

In this recipe, we configured a workspace in Terraform Cloud by adding the Azure environment variables configuration, which is an optional step and depends on the resources and cloud providers you wish to manage. In this workspace variables configuration, we can also add values for required variables or override default values. If, for example, in the configuration we have this variable:

```
variable "location" {
  type        = string
  description = " he location where resources will be deployed to."
}
```

We can set the value in the variables configuration of the workspace, as is shown in the following image:


Select variable category

● Terraform variable
These variables should match the declarations in your configuration. Click the HCL box to use interpolation or set a non-string value.

○ Environment variable
These variables are available in the Terraform runtime environment.

Key: location

Value: westeurope

☐ HCL ☐ Sensitive

Description (Optional): region of the resource

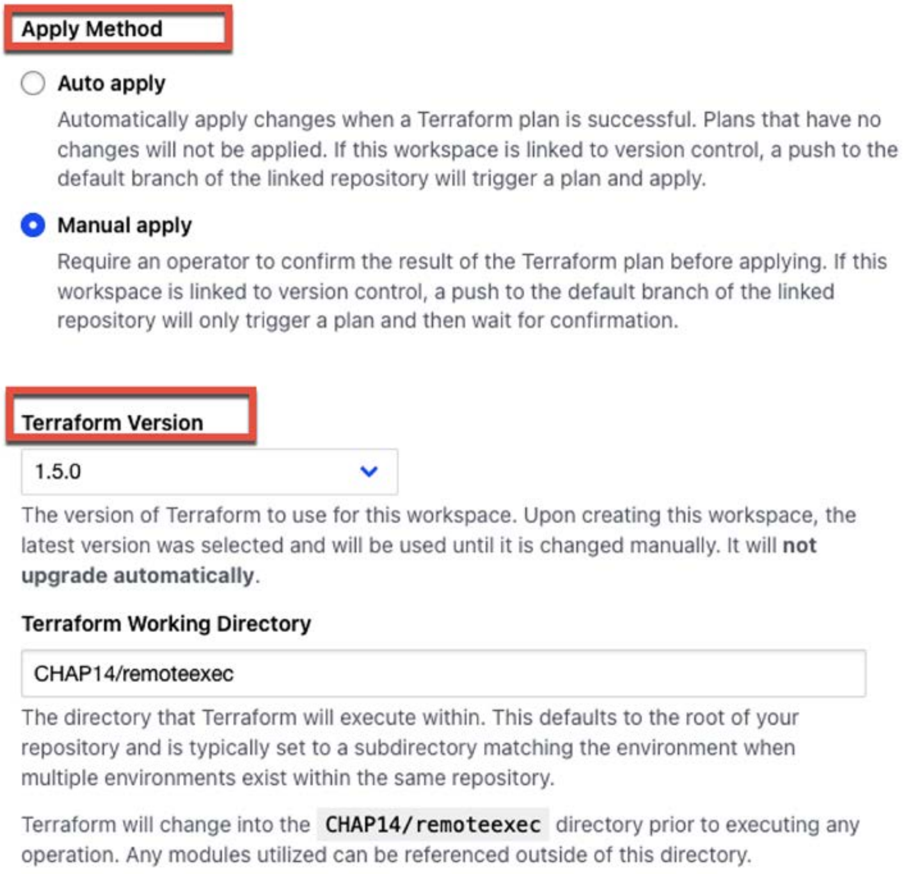
Figure 14.46: Adding a Terraform variable

Then, we run a remote execution of the Terraform configuration directly in Terraform Cloud with plan validation.

There's more...

In this recipe, we learned how to configure environment variables in the workspace settings. If we need to use the same environments in many workspaces, we can configure and use variable sets by following this tutorial: `https://developer.hashicorp.com/terraform/tutorials/cloud/cloud-multiple-variable-sets`.

We also learned how to run the `plan` and `apply` variables directly using the web interface of this platform. In the workspace settings, you can also configure whether you want to apply the plan manually (that is, with a confirmation, like in our recipe) or automatically. You can also choose the version of the Terraform binary you wish to use (by default, it uses the latest stable version that can be found at the time of the workspace's creation; beta versions are not considered):


Apply Method

[ ] Auto apply

Automatically apply changes when a Terraform plan is successful. Plans that have no changes will not be applied. If this workspace is linked to version control, a push to the default branch of the linked repository will trigger a plan and apply.

[✓ Manual apply]

Require an operator to confirm the result of the Terraform plan before applying. If this workspace is linked to version control, a push to the default branch of the linked repository will only trigger a plan and then wait for confirmation.

Terraform Version

1.5.0

The version of Terraform to use for this workspace. Upon creating this workspace, the latest version was selected and will be used until it is changed manually. It will **not upgrade automatically**.

Terraform Working Directory

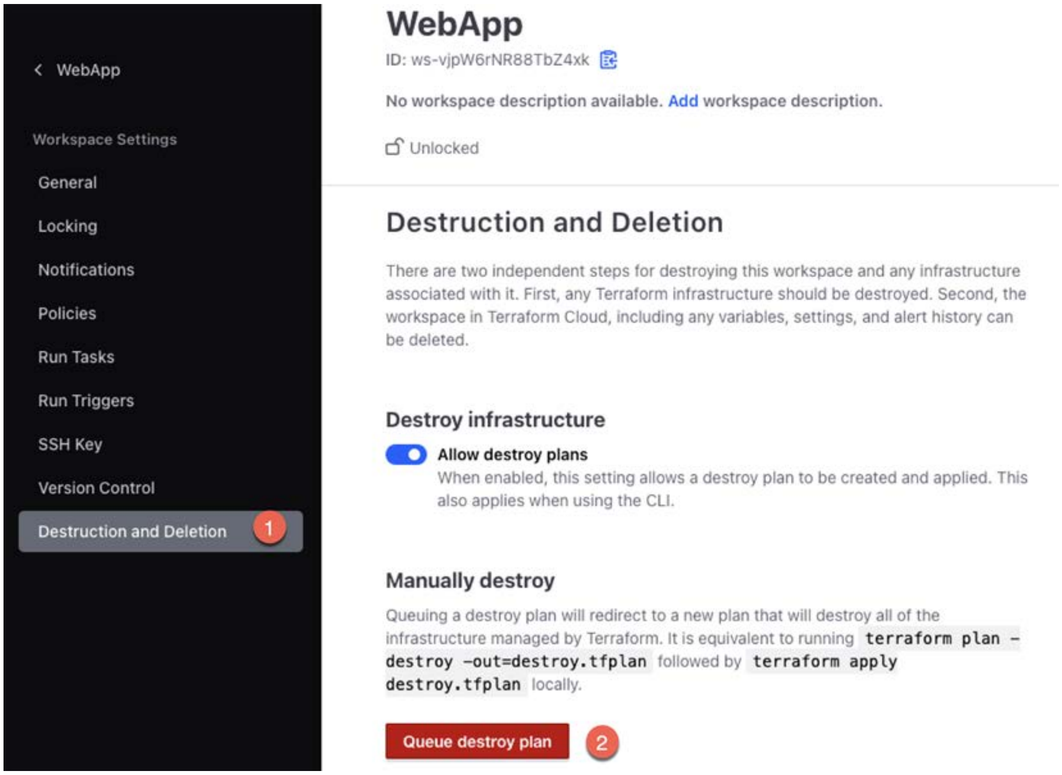
CHAP14/remoteexec

The directory that Terraform will execute within. This defaults to the root of your repository and is typically set to a subdirectory matching the environment when multiple environments exist within the same repository.

Terraform will change into the `CHAP14/remoteexec` directory prior to executing any operation. Any modules utilized can be referenced outside of this directory.

Figure 14.47: Specifying the Terraform version

You can also destroy all the resources that have been provisioned using the **Destruction and Deletion** feature, which is accessible by going to the workspace **Settings** > **Destruction and Deletion** menu and then clicking on the **Queue destroy plan** button:



WebApp

ID: ws-vjpW6rNR88TbZ4xk

No workspace description available. Add workspace description.

Unlocked

Destruction and Deletion

There are two independent steps for destroying this workspace and any infrastructure associated with it. First, any Terraform infrastructure should be destroyed. Second, the workspace in Terraform Cloud, including any variables, settings, and alert history can be deleted.

Destroy infrastructure

Allow destroy plans
When enabled, this setting allows a destroy plan to be created and applied. This also applies when using the CLI.

Manually destroy

Queuing a destroy plan will redirect to a new plan that will destroy all of the infrastructure managed by Terraform. It is equivalent to running `terraform plan -destroy -out=destroy.tfplan` followed by `terraform apply destroy.tfplan` locally.

Figure 14.48: TFC destroy operation

This operation will execute the `terraform destroy` command on the Terraform configuration.

In addition, as you may have noticed, by running the Terraform configuration in Terraform Cloud using the UI, we did not need to configure the `cloud` backend, as we discussed in the *Using a remote backend in Terraform Cloud* recipe of this chapter. In our case, the configuration of the Terraform state file is integrated with the workspace inside Terraform Cloud. One possible downside of this approach is that it may not be obvious to someone other than you, who is making changes to the configuration, where exactly the state is located and how to apply it. Keeping it alongside the configuration (i.e., declaring the cloud block) makes that much more obvious.

By clicking the **States** menu option, we'll notice the presence of the Terraform State:

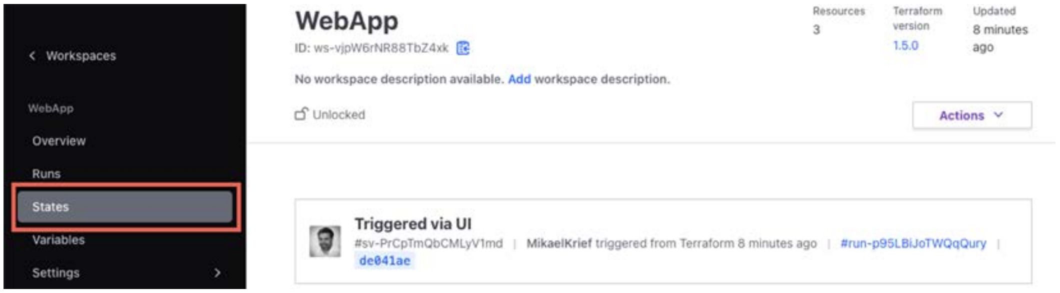
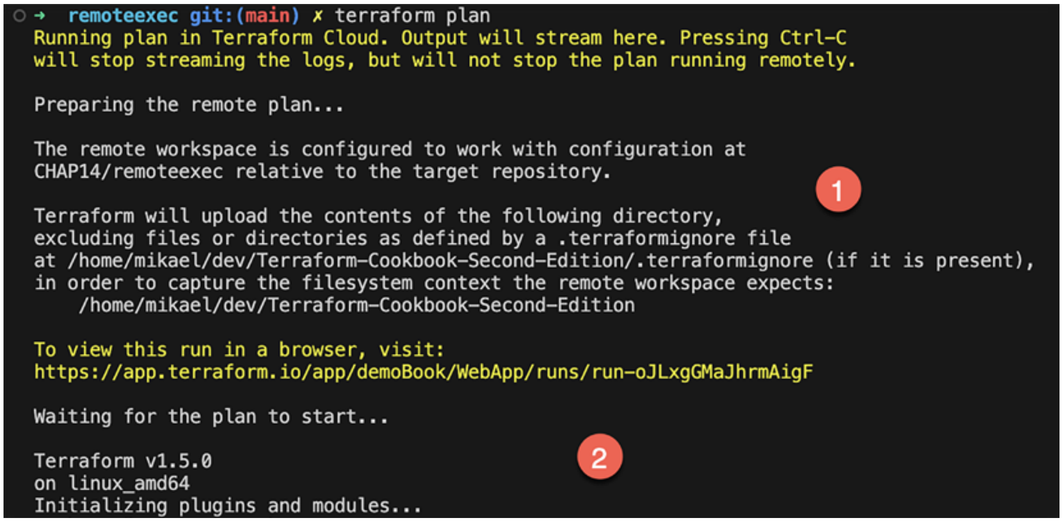


Figure 14.49: TFC state management

Moreover, if you are in a development context and want to check the development before committing to the repository, you can still use this remote mode of Terraform execution to make a `terraform plan`. This is done by controlling the execution that takes place in Terraform Cloud using your local Terraform CLI. To do this, simply add the configuration of the `cloud` backend, as in the *Using the remote backend in Terraform Cloud* recipe of this chapter, by using the name of the workspace we created in the first step of that recipe, which corresponds to the following code:

```
terraform {
  backend "remote" {
    hostname     = "app.terraform.io"
    organization = "demoBook"

    workspaces {
      name = "WebApp"
    }
  }
}
```

Then, on the development station, execute the `terraform plan` command, as shown in the following screenshot:



```
○ → remoteexec git:(main) ✗ terraform plan
Running plan in Terraform Cloud. Output will stream here. Pressing Ctrl-C
will stop streaming the logs, but will not stop the plan running remotely.

Preparing the remote plan...

The remote workspace is configured to work with configuration at
CHAP14/remoteexec relative to the target repository.

Terraform will upload the contents of the following directory,
excluding files or directories as defined by a .terraformignore file
at /home/mikael/dev/Terraform-Cookbook-Second-Edition/.terraformignore (if it is present),
in order to capture the filesystem context the remote workspace expects:
    /home/mikael/dev/Terraform-Cookbook-Second-Edition

To view this run in a browser, visit:
https://app.terraform.io/app/demoBook/WebApp/runs/run-oJLxgGMaJhrmAigF

Waiting for the plan to start...

Terraform v1.5.0
on linux_amd64
Initializing plugins and modules...
```

Figure 14.50: Migrating the state

During this execution, your Terraform CLI will create a configuration package and upload it to the Terraform Cloud workspace. The CLI triggers Terraform Cloud to run Terraform on the uploaded code. Finally, the output of the `plan` command is also available in the terminal. Please also note that, in this case, you do not need to set the environment variables locally since they are already configured in the workspace.

In order to ensure that the changes are applied in one place, you can't run the `apply` command on a workspace that is connected to a VCS. However, if your workspace is not connected to a VCS, then you can also execute the `apply` command from your local CLI.

Finally, if your Terraform configuration includes provisioning `local-exec` (which we studied in the *Executing local programs with Terraform* recipe in *Chapter 4, Using Terraform with External Data*) and, in its command, it uses a third-party tool, you will have to ensure that this tool is already present or install it on the Terraform Cloud agent, which will execute the Terraform binary. For more information about additional third-party tools in the execution of Terraform Cloud, I recommend reading the documentation available at `https://www.terraform.io/docs/cloud/run/install-software.html`.

See also

- The documentation on remote execution in Terraform Cloud is available here: `https://www.terraform.io/docs/cloud/run/index.html`.
- The documentation on using the CLI with remote execution is available here: `https://www.terraform.io/docs/cloud/run/cli.html`.

Checking the compliance of Terraform configurations using OPA in Terraform Cloud

The aspect of Terraform configuration tests was discussed in *Chapter 11*, *Running Test and Compliance Security on Terraform Configuration*, such as using **Open Policy Agent** (**OPA**), which was covered in the *Using Open Policy Agent for Terraform compliance* recipe.

In Terraform, compliance tests are carried out after the `terraform plan` command is executed. They verify that the result of the `plan` command corresponds to the rules described in the tests. Only if these tests have passed can the `terraform apply` command be executed.

Among the tools and frameworks for compliance testing, Terraform Cloud offers, in its free and paid plans, the stack, which allows us to write tests using the **Sentinel** or **OPA** framework and execute them directly in Terraform Cloud. This is done by using the run action between the `plan` and `apply` commands.

In this recipe, we will study a simple case of integrating OPA compliance tests and executing them in Terraform Cloud.

Getting ready

The essential requirement for this recipe is to have a Terraform Cloud organization.

Policies are rules that Terraform Cloud enforces on Terraform runs. You can define policies using either the Sentinel or OPA policy-as-code framework. Note that the Terraform Cloud Free edition includes one policy set of up to five policies. A policy set is a group of policies that may be enforced on workspaces.

> If you have an older Terraform Cloud organisation, you need to migrate to a new free plan as explained here: `https://www.hashicorp.com/blog/terraform-cloud-updates-plans-with-an-enhanced-free-tier-and-more-flexibility`

If you have the free plan, you can configure only one policy set per organization and you can't link your created policy set with a VCS repository.

Documentation about the plans, prices, and features is available here: `https://developer.hashicorp.com/terraform/cloud-docs/overview`.

The Terraform configuration used in this recipe is the one we already learned about in the recipe *Using Open Policy Agent for Terraform compliance* in *Chapter 11, Running Test and Compliance Security on Terraform Configuration*, available at `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/opa`. This configuration will create a Resource Group and an Azure Storage account.

In addition, to use the code written in this recipe in your TFC organization, you need to create a fork of the original repository of this book (`https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main`).

The goal of this recipe is to integrate the OPA rules, as part of policy sets, that will test the following: that the Azure Storage account is only available on HTTPS and that it is named `sademotestopa123`.

Then, we will learn how to apply these policy sets during the execution of Terraform Cloud.

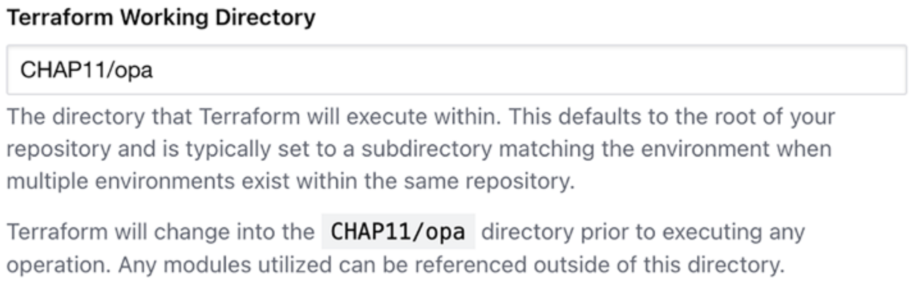
The purpose of this recipe is not to study all the elements for writing policies (for that, refer to the OPA documentation at `https://www.openpolicyagent.org/`). Here, we will be writing some simple code that you can easily reproduce.

Finally, the `storage` workspace must be created and configured in Terraform Cloud with a VCS provider, as described in the *Managing workspaces in Terraform Cloud* recipe of this chapter.

For this recipe, as we subscribed to the free plan of Terraform Cloud, we can't connect a policy set to our VCS provider. Therefore, we will create the policy directly in the Terraform Cloud UI.

The source code of the Terraform configuration used in this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP11/opa`.

The `storage` workspace is configured with the following setting:



Terraform Working Directory

CHAP11/opa

The directory that Terraform will execute within. This defaults to the root of your repository and is typically set to a subdirectory matching the environment when multiple environments exist within the same repository.

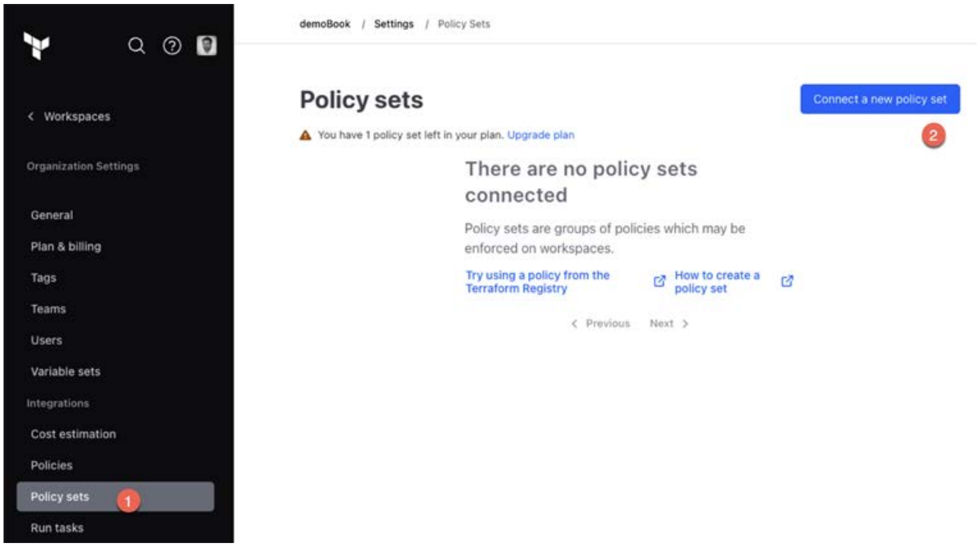
Terraform will change into the `CHAP11/opa` directory prior to executing any operation. Any modules utilized can be referenced outside of this directory.

Figure 14.51: TFC Terraform working directory

How to do it...

First, we need to configure policy sets in our Terraform Cloud organization:

1. In your organization **Settings**, select the **Policy sets** menu option and click on the **Connect a new policy set** button:



demoBook / Settings / Policy Sets

Policy sets

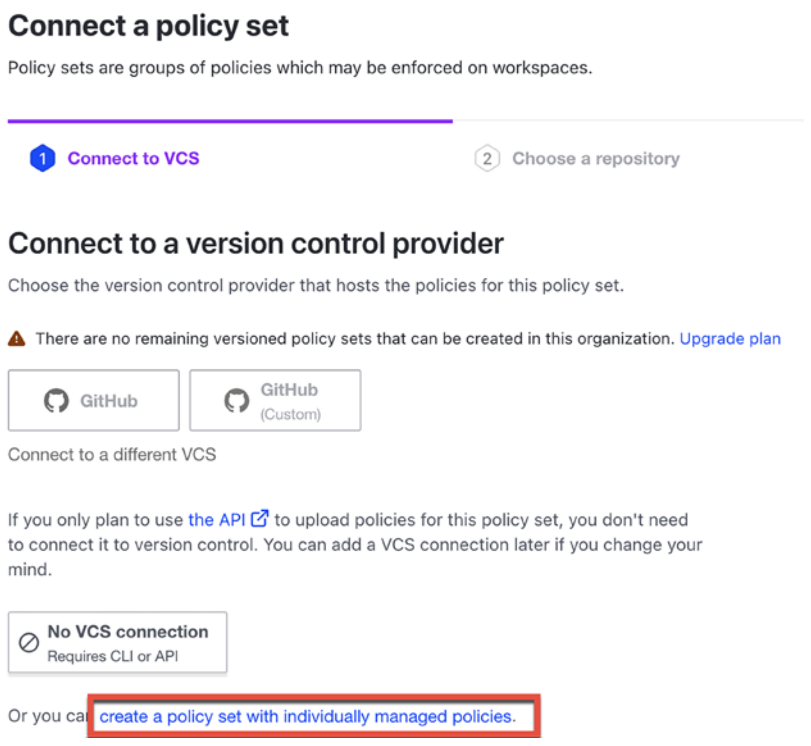
You have 1 policy set left in your plan. Upgrade plan

There are no policy sets connected

Policy sets are groups of policies which may be enforced on workspaces.

Figure 14.52: Creating a policy set

2. As we are using a free plan, we can't link the policy set to a VCS provider. So, we will click on the **create a policy set with individually managed policies** link, as shown in the following image:



Connect a policy set

Policy sets are groups of policies which may be enforced on workspaces.

1 Connect to VCS   2 Choose a repository

Connect to a version control provider

Choose the version control provider that hosts the policies for this policy set.

There are no remaining versioned policy sets that can be created in this organization. Upgrade plan

GitHub   GitHub (Custom)

Connect to a different VCS

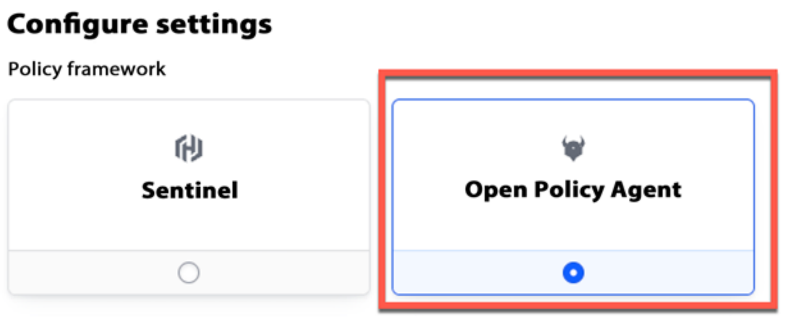
If you only plan to use the API to upload policies for this policy set, you don't need to connect it to version control. You can add a VCS connection later if you change your mind.

No VCS connection
Requires CLI or API

Or you can create a policy set with individually managed policies.

Figure 14.53: Connecting a policy set in TFC

3. In the newly opened form, configure the policy as follows:
   a. Choose **Open Policy Agent** as the policy framework.



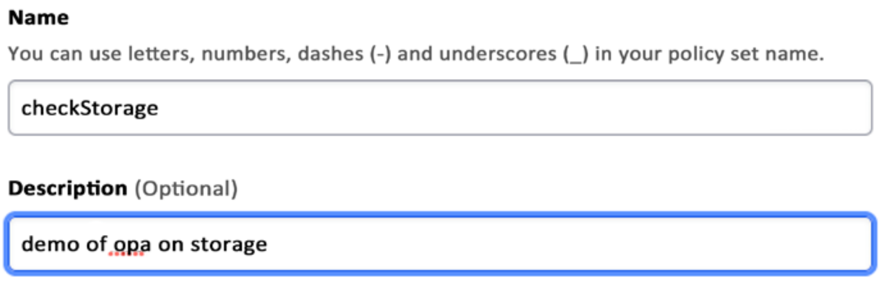
Configure settings

Policy framework

Sentinel ○   Open Policy Agent ●

Figure 14.54: Choosing the OPA framework

b. Name the policy set `checkStorage`.



Name

You can use letters, numbers, dashes (-) and underscores (_) in your policy set name.

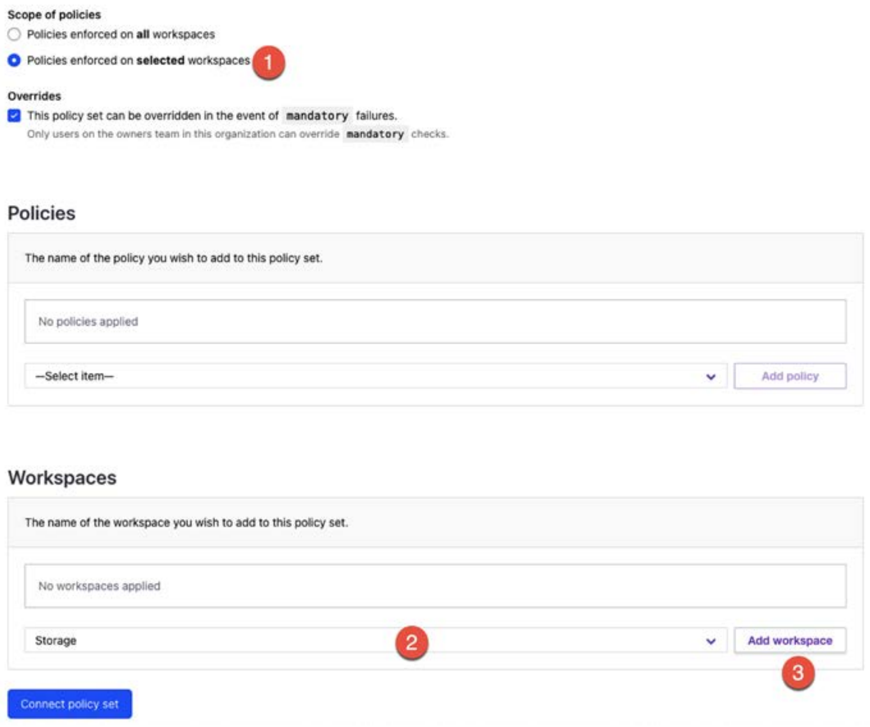
checkStorage

Description (Optional)

demo of opa on storage

Figure 14.55: Setting the policy name

c. Choose the scope of the policies as **selected workspaces** and select the **Storage** workspace from the available workspaces (leave the policies association empty).



Scope of policies

○ Policies enforced on **all** workspaces

● Policies enforced on **selected** workspaces

Overrides

☑ This policy set can be overridden in the event of `mandatory` failures.

Only users on the owners team in this organization can override `mandatory` checks.

Policies

The name of the policy you wish to add to this policy set.

No policies applied

—Select item—

Workspaces

The name of the workspace you wish to add to this policy set.

No workspaces applied

Storage

Figure 14.56: TFC policy configuration – select workspace

4. Finish by clicking on the **Connect policy set** button at the bottom of the above form.

Then, we will create the policy that will use OPA. Perform the following steps:

1. In the organization **Settings**, go to the **Policies** menu and click on the **Create a new policy** button.
2. Set the following configuration:
    a. Choose the **Open Policy Agent** policy framework.

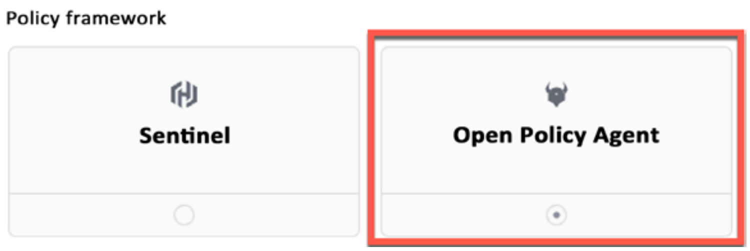
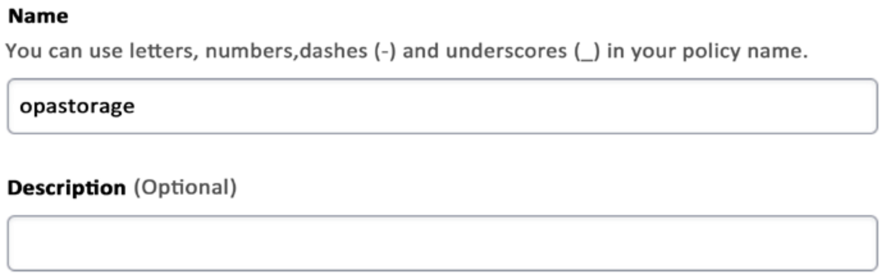


Figure 14.57: Selecting the OPA framework

    b. Enter the name of the policy.



Name

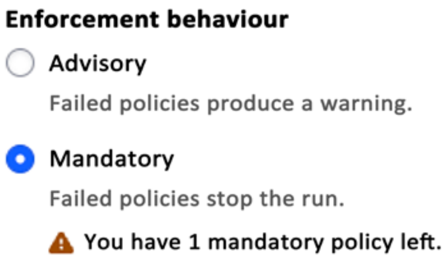
You can use letters, numbers,dashes (-) and underscores (_) in your policy name.

opastorage

Description (Optional)

Figure 14.58: Setting the policy name

    c. Set **Enforcement behavior** to **Mandatory**.



Enforcement behaviour

Advisory [✓ Mandatory]

Advisory: Failed policies produce a warning.

Mandatory: Failed policies stop the run.

You have 1 mandatory policy left.

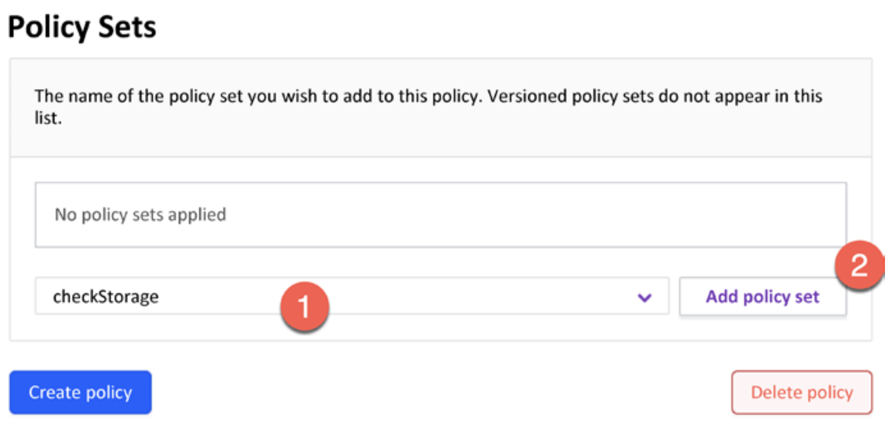
Figure 14.59: Setting the policy enforcement

d. In the **Policy code (Rego)** input box, write the Rego code of the OPA policy:

```
import input.plan as tfplan

azurerm_storage[resources] {
      resources := tfplan.resource_changes[_]
      resources.type == "azurerm_storage_account"
      resources.mode == "managed"
}

deny[msg] {
  az_storage := azurerm_storage[_]
  r := az_storage.change.after
  not r.enable_https_traffic_only
  msg := sprintf("Storage Account %v must use HTTPS traffic
only", [az_storage.name])
}

deny[msg] {
  az_storage := azurerm_storage[_]
  r := az_storage.change.after
  r.name != "sademotestopa123"
  msg := sprintf("Storage Account %v must be named
sademotestopa123", [az_storage.name])
}
```

The details of this code are explained in the recipe *Using Open Policy Agent for Terraform compliance* in *Chapter 11, Running Test and Compliance Security on Terraform Configuration*.

e. Select the policy set that this policy will be associated with. Choose the policy set **checkStorage** (created in the first part of this recipe):



Policy Sets

The name of the policy set you wish to add to this policy. Versioned policy sets do not appear in this list.

No policy sets applied

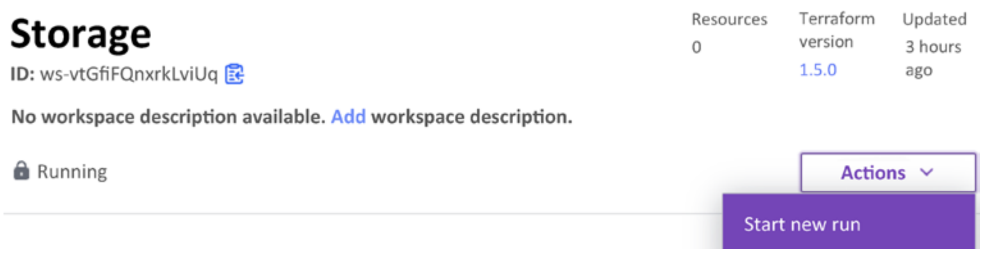
Figure 14.60: Associating a policy with a policy set

f. Create the policy by clicking on the **Create policy** button.

With this step, we have created a policy set that contains a policy that is configured to run on the **Storage** TFC workspace.

Now, the last part is to run the pipeline execution of the Storage workspace with the following steps:

1. In the Storage workspace, we click on the **Actions** > **Start new run** button (as we learned in detail in the *Executing Terraform configuration remotely in Terraform Cloud* recipe of this chapter):



Storage

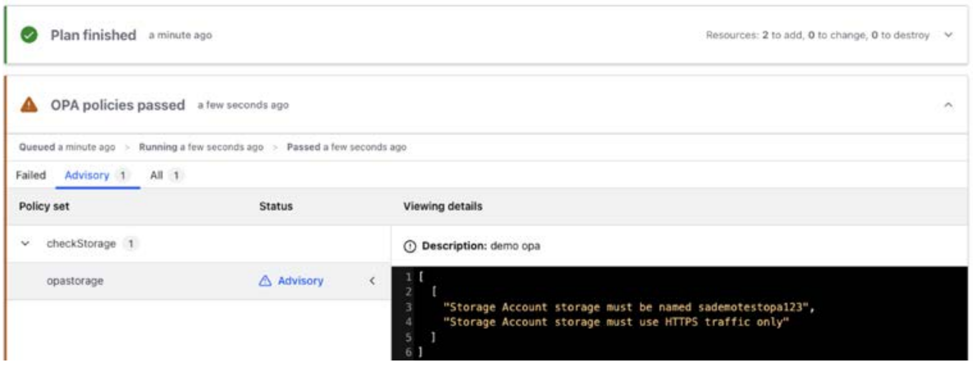
ID: ws-vtGfiFQnxrkLviUq

No workspace description available. Add workspace description.

Resources 0 | Terraform version 1.5.0 | Updated 3 hours ago

Figure 14.61: Start new run

2. In the details of the run execution, we can see the OPA execution just after the run plan:



Plan finished a minute ago Resources: 2 to add, 0 to change, 0 to destroy

OPA policies passed a few seconds ago

Queued a minute ago > Running a few seconds ago > Passed a few seconds ago

Failed Advisory 1 All 1

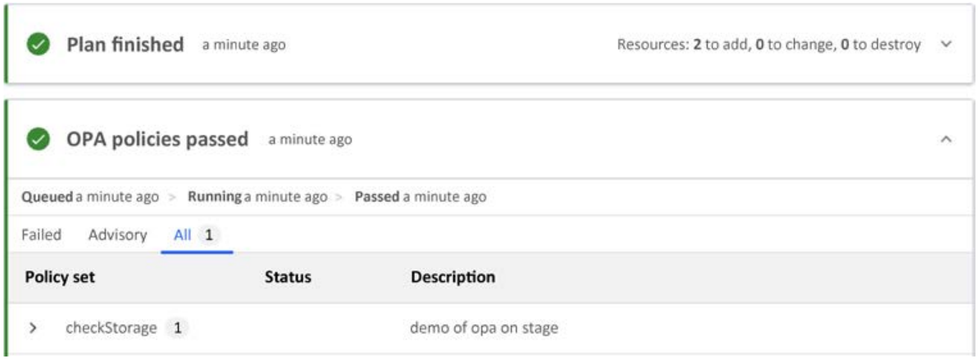
Policy set	Status	Viewing details
checkStorage 1		Description: demo opa
opastorage	Advisory	

```
[
  [
    "Storage Account storage must be named sademotestopa123",
    "Storage Account storage must use HTTPS traffic only"
  ]
]
```

Figure 14.62: OPA execution failed

In the above output, we can see the compliance of the HTTPS access and the name of the Azure Storage account.

3. Now, fix the compliance errors on Terraform configuration (read the recipe *Using Open Policy Agent for Terraform compliance* in *Chapter 11, Running Test and Compliance Security on Terraform Configuration*, for more details), commit the code to your GitHub repository, and rerun the workflow in Terraform Cloud. The following image shows the run result output:



Plan finished a minute ago Resources: 2 to add, 0 to change, 0 to destroy

OPA policies passed a minute ago

Queued a minute ago > Running a minute ago > Passed a minute ago

Failed Advisory All 1

Policy set	Status	Description
checkStorage 1		demo of opa on stage

Figure 14.63: OPA compliance passed successfully

The OPA compliance has passed successfully.

There's more...

In this recipe, we learned how to use the OPA framework to apply Terraform compliance in Terraform Cloud. We can also use the Sentinel framework, which I covered in the first edition of this book (but in this edition, I wanted to focus on OPA). For more information, refer to the following documentation resources:

- The code for Sentinel functions is available here: `https://github.com/hashicorp/terraform-guides/tree/master/governance/third-generation`.
- The documentation on how to install the Sentinel CLI is available at `https://docs.hashicorp.com/sentinel/intro/getting-started/install/`.
- The guide to writing and installing policies is available here: `https://www.hashicorp.com/resources/writing-and-testing-sentinel-policies-for-terraform/`.
- The basic learning guide for policies is available here: `https://learn.hashicorp.com/terraform/cloud-getting-started/enforce-policies`.
- Read this article to learn more about the use of Sentinel: `https://medium.com/hashicorp-engineering/using-new-sentinel-features-in-terraform-cloud-c1ade728cbb0`.
- The following is a video that demonstrates policy testing: `https://www.hashicorp.com/resources/testing-terraform-sentinel-policies-using-mocks/`.

See also

- A great HashiCorp tutorial on how to detect infrastructure drift and enforce OPA policies: `https://developer.hashicorp.com/terraform/tutorials/policy/drift-and-opa`.
- HashiCorp documentation on policy enforcement is available here: `https://developer.hashicorp.com/terraform/cloud-docs/policy-enforcement`.

Using integrated cost estimation for cloud resources

When we create resources in a cloud architecture, we often tend to forget that this incurs a financial cost that depends on the types of resources that are created. This is even more true with automation and IaC, which allow us to create a multitude of resources using a few commands.

One of the interesting features of Terraform Cloud is cost estimation, which makes it possible to visualize the cost of the resources that are handled in the Terraform configuration while it's being run.

In this recipe, we will learn how to use cost estimation in Terraform Cloud.

Getting ready

Before you start this recipe, you must have a Terraform Cloud organization and have created a project and workspace with VCS linked to a GitHub repository as we learned in the recipe *Managing workspaces in Terraform Cloud* in this chapter. Also, the execution mode of the workspace is configured as remote execution. For this recipe, the created workspace is called `MyApp`.

In Terraform Cloud, there are two solutions to manage cost infrastructure estimation:

- Using the cost estimation feature integrated into Terraform Cloud
- Using the Infracost run task

For more information on how to use the Infracost CLI, read the *Estimating the cost of Azure Infrastructure using Infracost* recipe in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*. For detailed implementation in Terraform Cloud, read the next recipe, *Integrating the Infracost Run task during the Terraform Cloud run*.

Note: There is a list of cloud providers and the resources that are supported by the cost management functionality available at `https://www.terraform.io/docs/cloud/cost-estimation/index.html#supported-resources`.

The purpose of this recipe is to provision an Azure virtual machine and Azure App Service and to visualize the cost estimation of these resources in the Terraform Cloud interface using the cost integration feature.

The source code for the Terraform configuration that will be executed in this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP14/costestimation`.

How to do it…

To view the estimation of costs for our resources using integrated cost management, perform the following steps:

1. In the **Settings** section of the Terraform Cloud organization, in the **Cost Estimation** tab, check the **Enable Cost Estimation for all workspaces** checkbox and then click on the **Update settings** button, as shown in the following image:

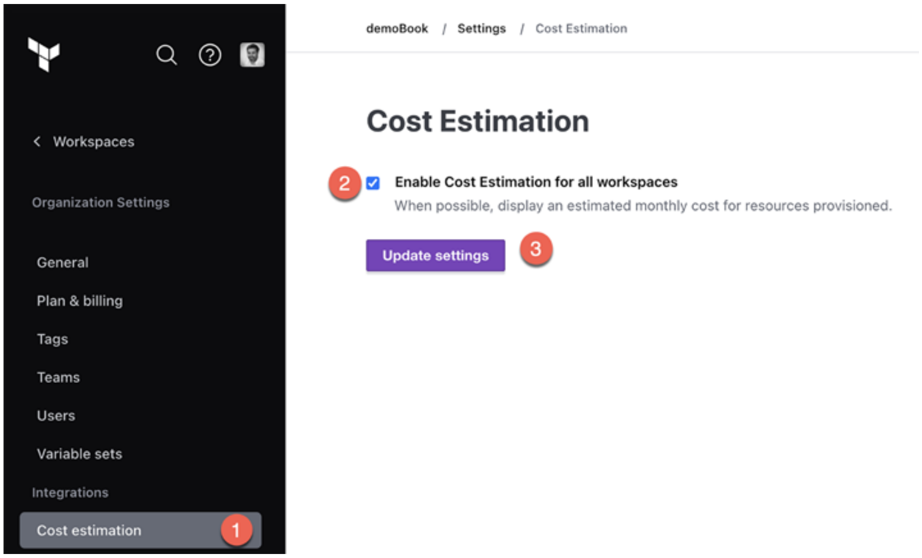

demoBook / Settings / Cost Estimation
Workspaces
Organization Settings
General
Plan & billing
Tags
Teams
Users
Variable sets
Integrations
Cost estimation
1
Cost Estimation
2
Enable Cost Estimation for all workspaces
When possible, display an estimated monthly cost for resources provisioned.
Update settings
3

Figure 14.64: Enabling integrated Cost Estimation

2. In the workspace used, `MyApp`, which provisions the Azure resources with our Terraform configuration, click on the **Actions** button and then click on **Start new run**, as shown in the following image:

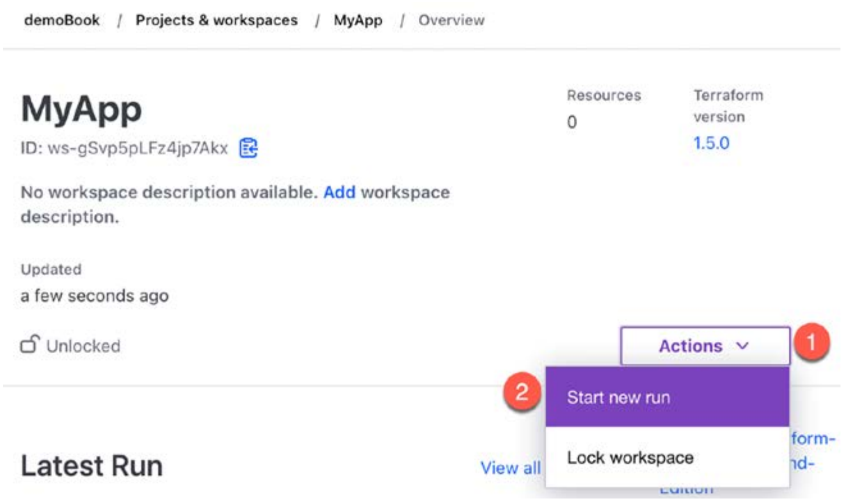
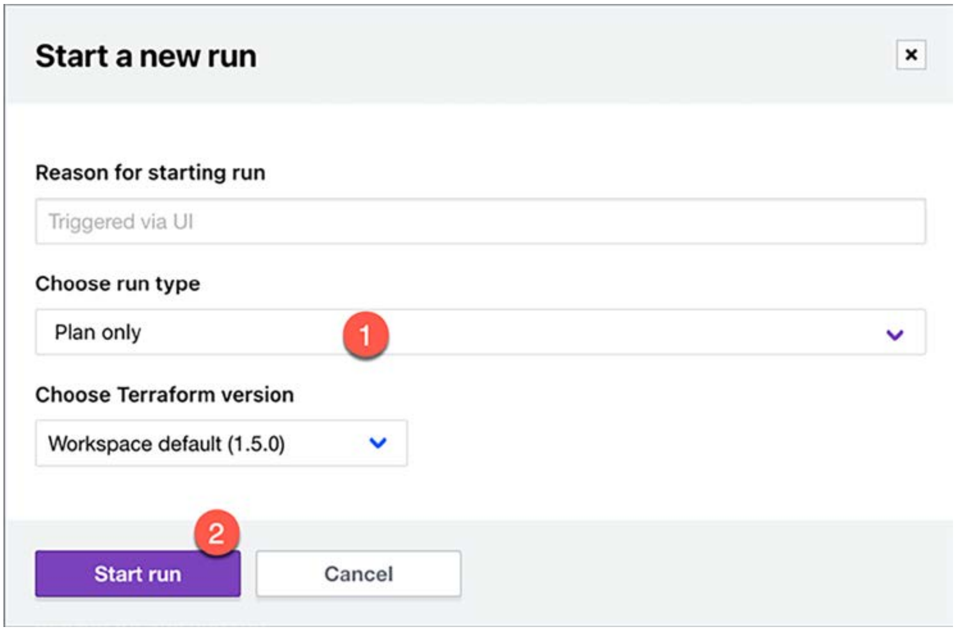


Figure 14.65: Start new run

3. For the provider's authentication, here for Azure, in the **Variables** tab set your four Azure authentication variables as explained in the *Executing Terraform configuration remotely in Terraform Cloud* recipe in this chapter.
4. In the window that opens, choose the option to run only the plan (labeled 1), and click on **Start run**:


Start a new run

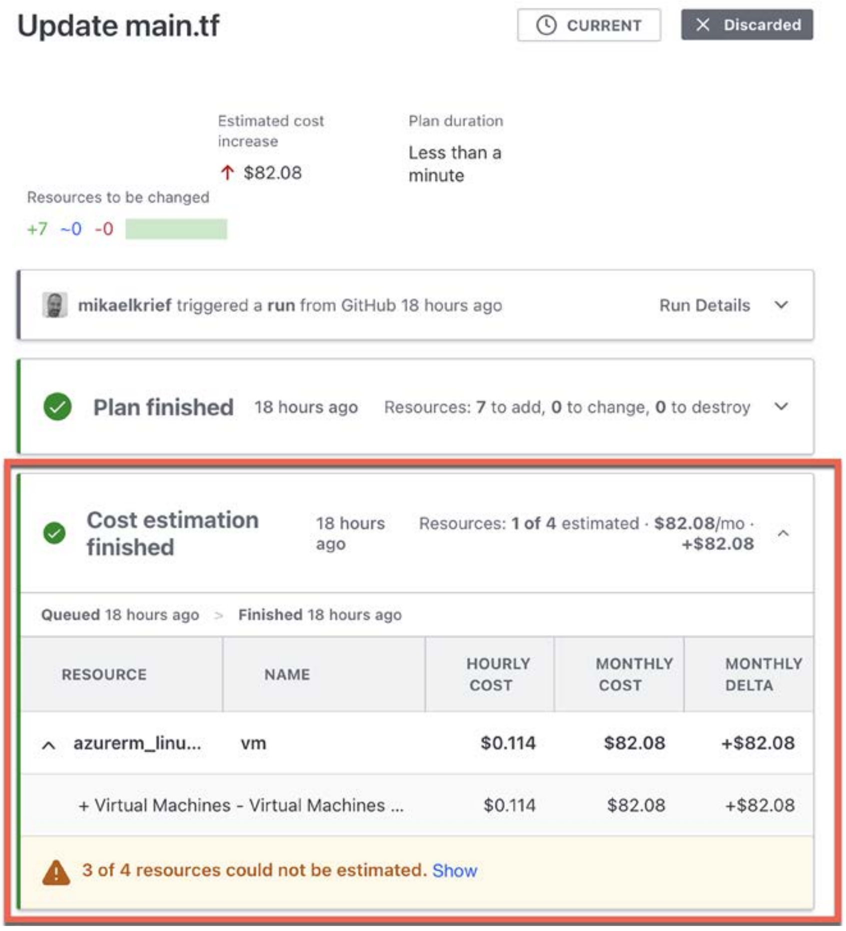
Reason for starting run: Triggered via UI

Choose run type: Plan only

Choose Terraform version: Workspace default (1.5.0)

Figure 14.66: Start run

5. After executing our plan, click on the **See details** button on the run. We can view the evaluated cost of the resources:



Update main.tf

Estimated cost increase ↑ $82.08

Plan duration Less than a minute

Resources to be changed +7 ~0 -0

mikaelkrief triggered a run from GitHub 18 hours ago

Plan finished 18 hours ago Resources: 7 to add, 0 to change, 0 to destroy

Cost estimation finished 18 hours ago Resources: 1 of 4 estimated · $82.08/mo · +$82.08

Queued 18 hours ago > Finished 18 hours ago

RESOURCE	NAME	HOURLY COST	MONTHLY COST	MONTHLY DELTA
azurerm_linu...	vm	$0.114	$82.08	+$82.08
+ Virtual Machines - Virtual Machines ...		$0.114	$82.08	+$82.08

3 of 4 resources could not be estimated. Show

Figure 14.67: Integrated cost estimation during run

We can see the cost estimation details in the above image output.

How it works...

Once the cost estimation option has been activated, Terraform Cloud uses the APIs of the different cloud providers to evaluate and display the costs of the resources that will be provisioned.

There's more...

It is important to note that this is only an estimate and that it is necessary to refer to the different price documentation of the cloud providers.

Another thing to be aware of is that activating the cost estimation feature will estimate costs for all workspaces and organizations. You can also write policies with Sentinel (which we studied in the previous recipe) to integrate compliance rules for estimated costs. For more information, please read the documentation at `https://www.terraform.io/docs/cloud/cost-estimation/index.html#verifying-costs-in-policies`.

See also

- The documentation regarding the integrated cost estimation feature is available here: `https://www.terraform.io/docs/cloud/cost-estimation/index.html`.

Integrating the Infracost run task during the Terraform Cloud run

In the previous recipe, we learned how to estimate cost infrastructure using the integrated cost estimation feature in Terraform Cloud.

In this recipe, we will learn how to execute a run task during the run pipeline and we will use the Infracost run task as an example.

Terraform Cloud run tasks are third-party tools and services that allow you to perform some custom operations during the Terraform workflow. To get the list of all run tasks already integrated into the Terraform Cloud registry, look here: `https://registry.terraform.io/browse/run-tasks?product_intent=terraform`.

In this book, I will not detail everything regarding the addition and configuration of all run task integrations. For more details, refer to the documentation here: `https://developer.hashicorp.com/terraform/cloud-docs/workspaces/settings/run-tasks`. In this recipe, we will learn about the usage of the Infracost run task.

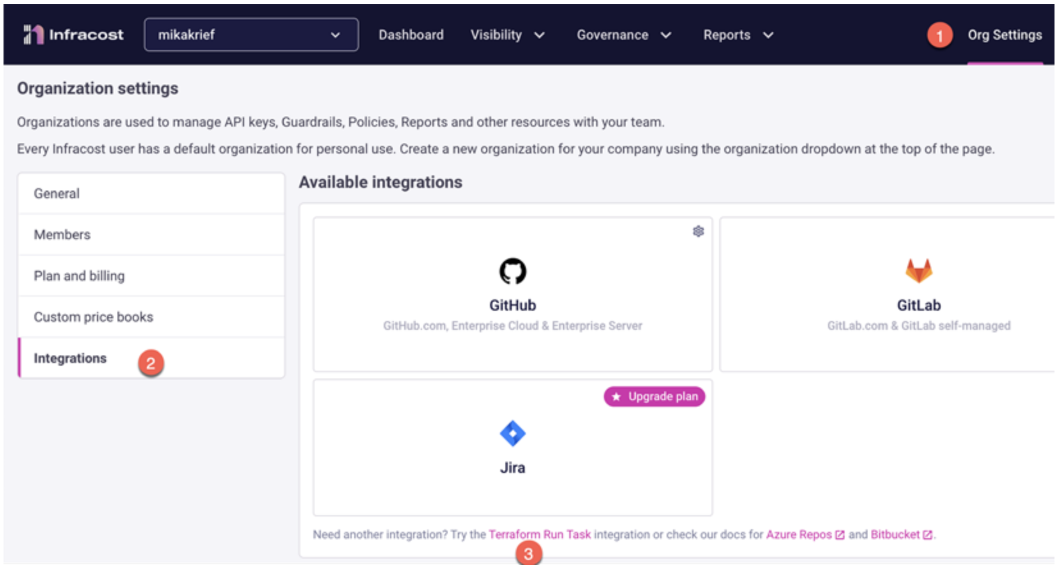
Getting ready

Before you start this recipe, you must have a Terraform Cloud subscription and have created one project and one workspace with VCS configured and linked to the GitHub repository, as we learned in the *Managing workspaces in Terraform Cloud* recipe in this chapter. Also, the workspace is configured as remote execution. For this recipe, the created workspace is called `MyApp`.

In this recipe, we will use the Infracost run task (for more information on how to use the Infracost CLI, read the *Estimating the cost of Azure infrastructure using Infracost* recipe in *Chapter 8*, *Provisioning Azure Infrastructure with Terraform*).

Before using the Infracost run task (available here: `https://registry.terraform.io/browse/run-tasks?category=cost-management`), you need to create an account on Infracost here: `https://dashboard.infracost.io/`. Also refer to the list of supported cloud providers here: `https://www.infracost.io/docs/supported_resources/overview/`.

Additionally, to use Infracost in Terraform Cloud, we need to get the Infracost **Endpoint URL** and the **HMAC key**. For this, we log in to our Infracost account, click on **Org settings** > **Integrations**, and click on the link **Terraform Run Task**, as shown in the following image:



Organization settings

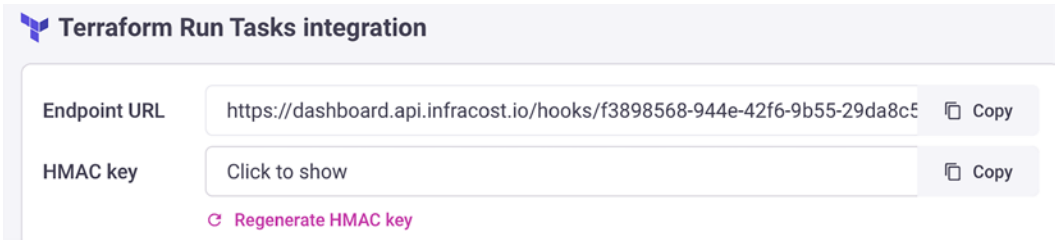
Organizations are used to manage API keys, Guardrails, Policies, Reports and other resources with your team.

Every Infracost user has a default organization for personal use. Create a new organization for your company using the organization dropdown at the top of the page.

Need another integration? Try the Terraform Run Task integration or check our docs for Azure Repos and Bitbucket.

Figure 14.68: Infracost TFC integration

Then get the provided **Endpoint URL** and **HMAC key** values.

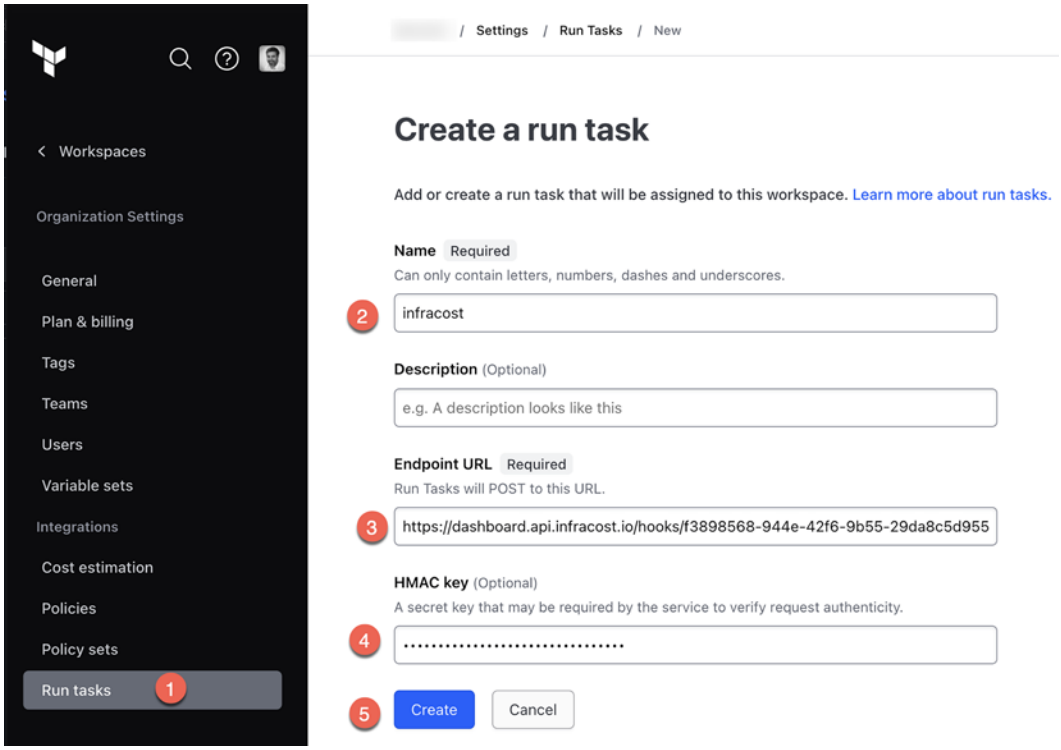

Terraform Run Tasks integration

Endpoint URL: https://dashboard.api.infracost.io/hooks/f3898568-944e-42f6-9b55-29da8c5

HMAC key: Click to show

Figure 14.69: Infracost Endpoint URL and HMAC key

Finally, go to the Terraform Cloud organization and click on the **Settings** > **Run tasks** > **Create run tasks** menu, and fill in the form with the name, the endpoint URL, and the HMAC of the run task as shown in the following image:


Create a run task

Add or create a run task that will be assigned to this workspace. Learn more about run tasks.

Name Required
Can only contain letters, numbers, dashes and underscores.
infracost

Description (Optional)
e.g. A description looks like this

Endpoint URL Required
Run Tasks will POST to this URL.
https://dashboard.api.infracost.io/hooks/f3898568-944e-42f6-9b55-29da8c5d955

HMAC key (Optional)
A secret key that may be required by the service to verify request authenticity.
••••••••••••••••••••••••••••••••

Figure 14.70: Creating an Infracost run task

Now we have configured the Infracost run task.

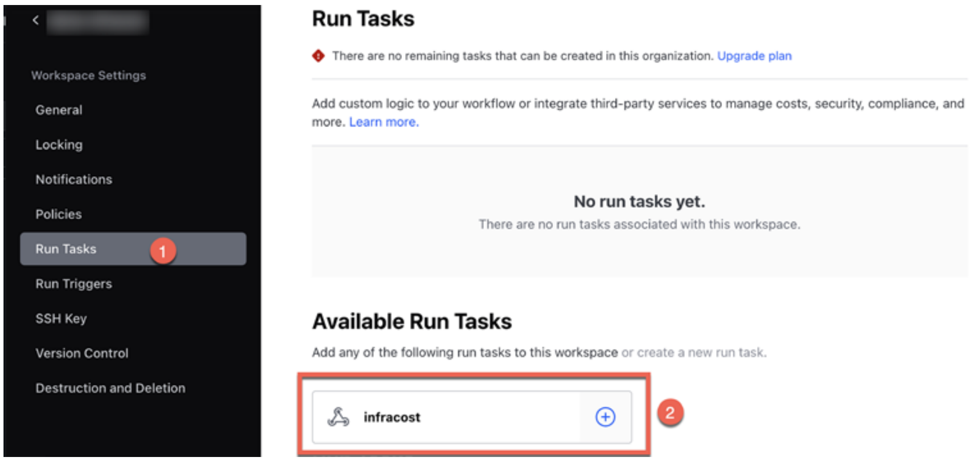
The purpose of this recipe is to provision an Azure virtual machine and Azure Web Apps and to visualize the cost estimation of these resources in the Terraform Cloud interface using Infracost.

The source code for the Terraform configuration that will be executed in this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP14/costestimation`.

How to do it…

To estimate costs in Terraform Cloud using the Infracost run task, perform the following steps:

1. Go to the **Settings** of the Terraform Cloud workspace `MyApp` and inside the **Run Tasks** configuration, choose to use the Infracost run task that we configured in the requirements of this recipe.



There are no remaining tasks that can be created in this organization.

Add custom logic to your workflow or integrate third-party services to manage costs, security, compliance, and more.

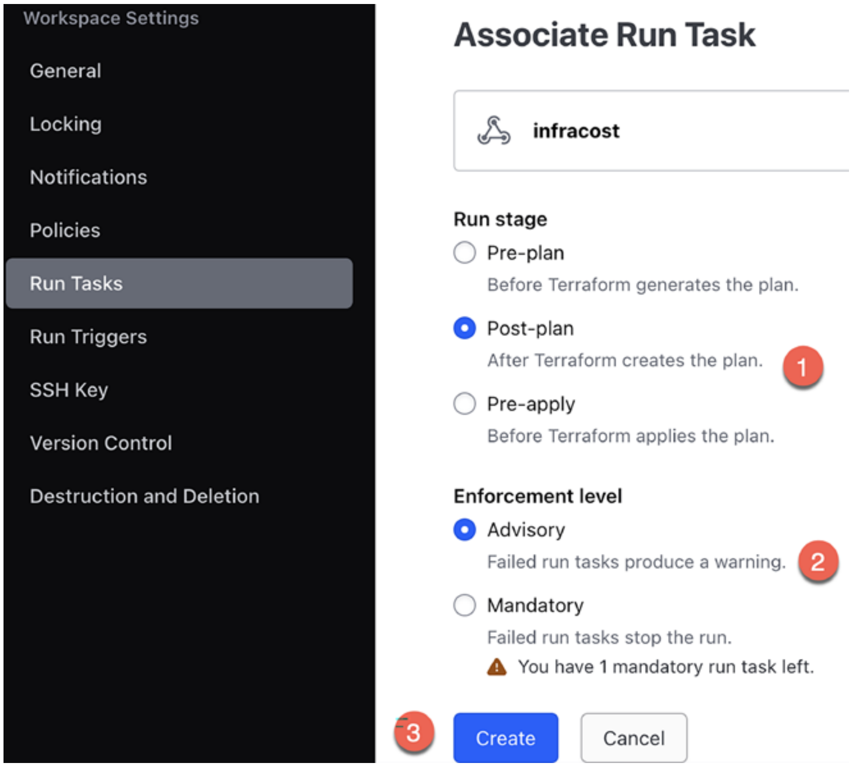
No run tasks yet.

There are no run tasks associated with this workspace.

Add any of the following run tasks to this workspace or create a new run task.

Figure 14.71: Infracost run task

2. Then, in the next form, we keep the default configuration to run the Infracost run task in the **Post-plan** stage (just after the execution of `terraform plan`) and keep the enforcement level as **Advisory**.



Associate Run Task

infracost

Run stage

( ) Pre-plan
Before Terraform generates the plan.

(●) Post-plan
After Terraform creates the plan.

( ) Pre-apply
Before Terraform applies the plan.

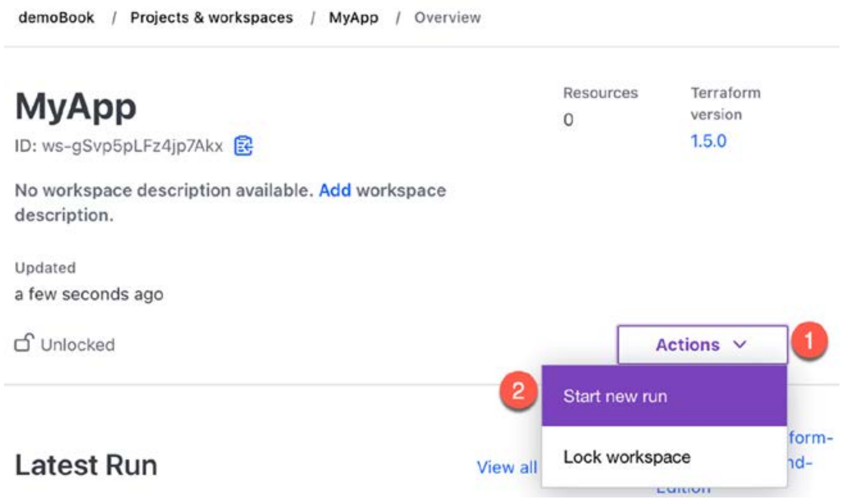
Enforcement level

(●) Advisory
Failed run tasks produce a warning.

( ) Mandatory
Failed run tasks stop the run.
⚠ You have 1 mandatory run task left.

Figure 14.72: Configuration of the Infracost run task

3. In the workspace `MyApp`, which provisions the Azure resources with our Terraform configuration, click on the **Actions** button and then on **Start new run**, as shown in the following image:


demoBook / Projects & workspaces / MyApp / Overview

MyApp
ID: ws-gSvp5pLFz4jp7Akx

Resources
0

Terraform version
1.5.0

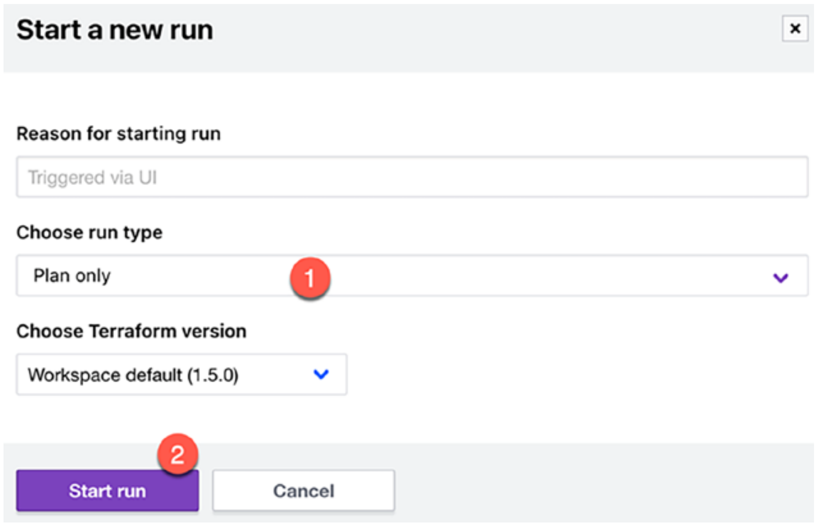
No workspace description available. Add workspace description.

Updated
a few seconds ago

Unlocked

Figure 14.73: Start new run

4. In the window that opens, choose the option to run only the plan, and click on **Start run**.


Start a new run

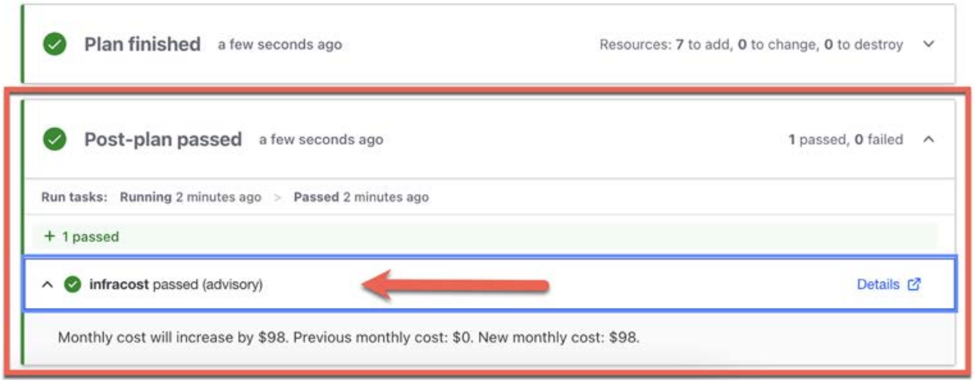
Reason for starting run
Triggered via UI

Choose run type
Plan only

Choose Terraform version
Workspace default (1.5.0)

Figure 14.74: Start run

5. In the details of the run, we can see the post-plan execution with the Infracost run task execution and the cost estimation summary:



Plan finished a few seconds ago Resources: 7 to add, 0 to change, 0 to destroy

Post-plan passed a few seconds ago 1 passed, 0 failed

Run tasks: Running 2 minutes ago > Passed 2 minutes ago

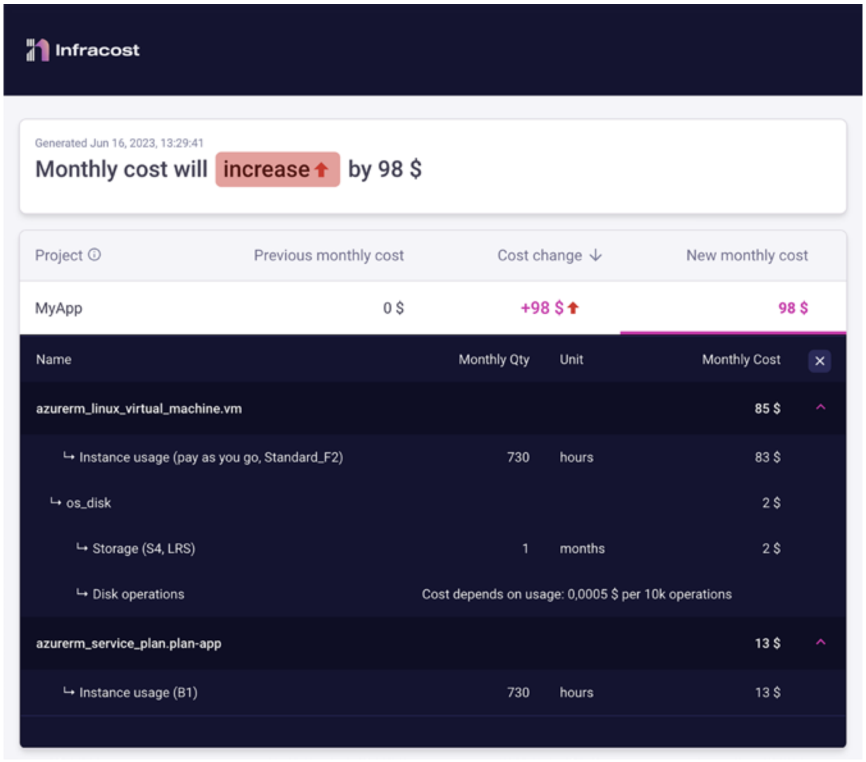
+ 1 passed

infracost passed (advisory) Details

Monthly cost will increase by $98. Previous monthly cost: $0. New monthly cost: $98.

Figure 14.75: TFC run details with Infracost execution

6. For more details of this cost estimation, click on the **Details** link (on the **Infracost** row), which will open a new web page on your Infracost cloud dashboard.



Infracost

Generated Jun 16, 2023, 13:29:41

Monthly cost will increase by 98 $

Project	Previous monthly cost	Cost change	New monthly cost
MyApp	0 $	+98 $	98 $

Name	Monthly Qty	Unit	Monthly Cost
azurerm_linux_virtual_machine.vm			85 $
↳ Instance usage (pay as you go, Standard_F2)	730	hours	83 $
↳ os_disk			2 $
↳ Storage (S4, LRS)	1	months	2 $
↳ Disk operations	Cost depends on usage: 0,0005 $ per 10k operations		
azurerm_service_plan.plan-app			13 $
↳ Instance usage (B1)	730	hours	13 $

Figure 14.76: Infracost cost estimation details

We can see in this image the detailed cost estimation performed by Infracost.

How it works...

Once the Infracost run task has been activated, Infracost uses the APIs of the different cloud providers to evaluate and display the costs of the resources that will be provisioned.

There's more...

It is important to note that this is only an estimate and that it is necessary to refer to the different price documentation of the cloud providers.

See also

- The Infracost documentation is available here: `https://www.infracost.io/`.
- The Infracost integration with Terraform Cloud documentation is available here: `https://www.infracost.io/docs/integrations/terraform_cloud_enterprise/`.

Configuring Terraform Cloud with the Terraform TFE provider

In the previous recipes of this chapter, we learned how to use projects, workspaces, and run tasks in Terraform Cloud using the web interface. In a company context, with large-scale infrastructure and a lot of Terraform Cloud organizations and workspaces, the manual actions to configure all of them can be difficult to maintain and need to be automatized.

To automate the configuration of Terraform Cloud, we can use a Terraform provider called TFE.

The TFE provider is specifically designed to configure and manage resources within Terraform Cloud (as well as Terraform Enterprise).

Here are a few reasons why you would use the TFE provider to configure Terraform Cloud:

- **Integration with Terraform Cloud**: The TFE provider allows you to interact with and configure Terraform Cloud resources directly from your Terraform configuration files. It provides a set of resources and data sources that map to various components and functionalities of Terraform Cloud, such as workspaces, variables, state management, and runs.
- **Automation and infrastructure as code**: By using the TFE provider, you can automate the configuration of Terraform Cloud resources, ensuring consistent and reproducible workflows. It enables you to define and manage your Terraform Cloud infrastructure as code, making it easier to version control, share, and collaborate on infrastructure configurations.

- **Collaboration and teamwork**: Terraform Cloud offers features like team management, access controls, and collaboration tools that facilitate working with multiple team members on infrastructure provisioning. The TFE provider allows you to automate the setup and management of teams, user access permissions, and other collaboration aspects within Terraform Cloud, streamlining the teamwork process.
- **Enhanced state management**: Terraform Cloud provides centralized state storage, which enables multiple team members to work on the same infrastructure codebase concurrently. The TFE provider allows you to configure and manage state storage configurations, including remote backends and state locking, ensuring safe and efficient state management across your organization.
- **Auditing and governance**: Terraform Cloud offers audit logging and governance features, allowing you to track changes, review logs, and enforce policy checks. The TFE provider enables you to configure these auditing and governance settings, ensuring compliance with organizational requirements and providing visibility into infrastructure changes.

Overall, using the TFE provider simplifies the configuration and management of Terraform Cloud resources, automates workflows, and enhances collaboration and governance capabilities, making it an effective choice for provisioning and managing infrastructure in Terraform Cloud.

In this recipe, we will see an example of Terraform configuration to configure Terraform Cloud using the TFE Terraform provider.

Let’s get started!

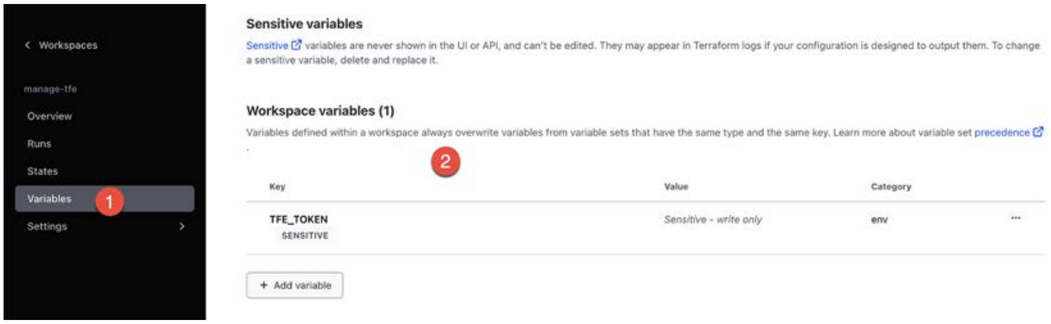
Getting ready

To complete this recipe, you’ll need to have an existing Terraform Cloud organization, which will be the “main” organization (which we named `demo-tfe` in this example), and a workspace named `main-tfe`, which will just store the Terraform State of the Terraform configuration that will run in the recipe.

To gain permission to create other organizations, one method is to authenticate to Terraform Cloud in the terminal console that will run the Terraform commands. To authenticate, use the `terraform login` command or create a new Terraform Cloud user token, as we learned in the *Authenticating Terraform to Terraform Cloud* recipe of this chapter.

For more details about the TFE authentication, refer to the documentation here: `https://registry.terraform.io/providers/hashicorp/tfe/latest/docs#authentication`.

In the `manage-tfe` workspace in the **Variables** configuration, we will set a new variable as an environment variable that contains the value of the user token:



Sensitive variables

Sensitive variables are never shown in the UI or API, and can't be edited. They may appear in Terraform logs if your configuration is designed to output them. To change a sensitive variable, delete and replace it.

Workspace variables (1)

Variables defined within a workspace always overwrite variables from variable sets that have the same type and the same key. Learn more about variable set precedence

Key	Value	Category
TFE_TOKEN SENSITIVE	Sensitive - write only	env

Figure 14.77: TFC TFE_TOKEN environment variable configuration

The goal of this recipe is to create a new Terraform Cloud organization, a project, and, inside this project, two workspaces using Terraform configuration and the TFE provider.

How to do it…

To use the TFE provider, perform the following steps:

1. In the new `main.tf` file, add the following configuration:

```
terraform {
  required_version = "~> 1.1.0"
  required_providers {
    tfe = {
      source  = "hashicorp/tfe"
      version = "0.45.0"
    }
  }
  cloud {
    hostname     = "app.terraform.io"
    organization = "demo-tfe"

    workspaces {
      name = "manage-tfe"
    }
  }
}
```

In this configuration, we add the `tfe` provider and configure a remote backend on Terraform Cloud in the `manage-tfe` workspace.

2. Then, add the following Terraform configuration:

```
provider "tfe" {}

resource "tfe_organization" "test-organization" {
  name  = "demo-tfe-book-${random_string.random.result}"
  email = "demo@tfcookobook.com"
}

resource "tfe_project" "project" {
  organization = tfe_organization.test-organization.name
  name         = "appproject"
}

resource "tfe_workspace" "wsnetwork" {
  name         = "network"
  organization = tfe_organization.test-organization.name
}

resource "tfe_workspace" "wsvm" {
  name         = "vm"
  organization = tfe_organization.test-organization.name
}
```

This Terraform configuration creates one organization, `demo-tfe-book`, one project, `appproject`, and two workspaces, `wsnetwork` and `wsvm`.

3. Finally, in the terminal console, run the basic Terraform workflow with the `init`, `plan`, and `apply` commands.

4. Check the creation of the Terraform Cloud resources:

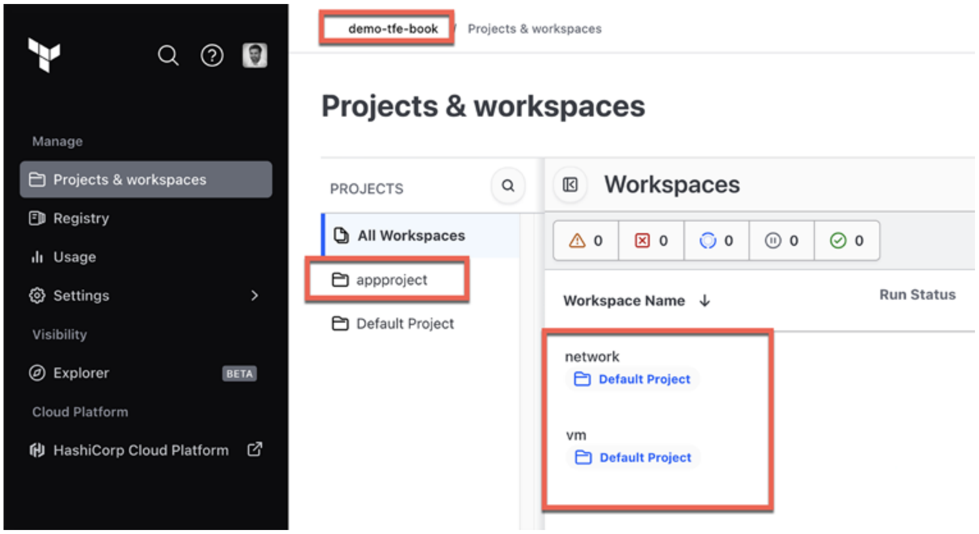


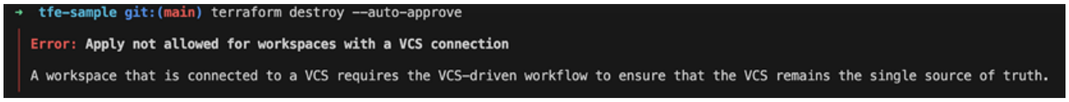
Figure 14.78: TFC project and workspace created by the TFE provider

We can see that TFE creates the `demo-tfe-book` organization, the project `appproject`, and the two workspaces, `network` and `vm`.

There's more…

In this recipe, in *Step 3*, we executed a Terraform workflow that was run inside TFC, and the output was streamed back into the terminal. We can also run the pipeline directly in Terraform Cloud by connecting the `manage-tfe` workspace with the VCS that contains this Terraform configuration, as we learned in detail in the *Executing Terraform configuration remotely in Terraform Cloud* recipe in this chapter.

But once we have configured the VCS in the workspace, we can't run Terraform commands via the terminal console as shown in the following image:



```
→  tfe-sample git:(main) terraform destroy --auto-approve
│ Error: Apply not allowed for workspaces with a VCS connection
│ A workspace that is connected to a VCS requires the VCS-driven workflow to ensure that the VCS remains the single source of truth.
```

Figure 14.79: Terraform destroy error when attempting TFC integration

We can see the error on running the `destroy` command, as it doesn't allow running Terraform commands in the terminal.

See also

- The TFE documentation is available here: `https://registry.terraform.io/providers/hashicorp/tfe/latest/docs`.
- A case study on the use of TFE can be found at `https://www.hashicorp.com/resources/managing-workspaces-with-the-tfe-provider-at-scale-factory`.
- Article on TFE usage (by Ned Bellavance): `https://nedinthecloud.com/2022/02/03/managing-terraform-cloud-with-the-tfe-provider/`.
- GitHub source code of HashiCorp tutorial on TFE: `https://github.com/hashicorp/learn-terraform-tfe-provider-run-triggers/`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

15

Troubleshooting Terraform Errors

We've now reached the final chapter of this book. Throughout this book, we've learned how to use Terraform to provision infrastructure resources. As you learn more about Terraform, you may come across Terraform configuration errors that will require troubleshooting.

In this chapter, we'll look at how to debug and fix some of the common errors identified by HashiCorp in the documentation here: `https://developer.hashicorp.com/terraform/tutorials/configuration-language/troubleshooting-workflow`.

We'll look at how to correct interpolation, the address cycle, `for_each` loops, and output errors. In each of the recipes, we'll look at errors in the Terraform configuration, the steps for reproducing the errors, and finally, the steps for correcting the errors.

The chapter doesn't cover all types of Terraform errors. There are other errors (some of which we have already studied in this book), for example:

- The locking or corruption of the Terraform state, with more information here: `https://www.pluralsight.com/resources/blog/cloud/how-to-troubleshoot-5-common-terraform-errors#h-5-terraform-state-errors`.
- The *resource already exists* error; see this article for more information: `https://saturncloud.io/blog/getting-started-with-terraform-resolving-resource-already-exists-error-with-newly-created-resource/` and the *Importing existing resources* recipe of *Chapter 5*, *Managing Terraform State*.

This chapter covers:

- Fixing interpolation errors
- Fixing cycle errors
- Fixing `for_each` errors
- Fixing output errors

Fixing interpolation errors

Terraform interpolation is a feature provided by Terraform that allows you to embed expressions within strings or configuration blocks. These expressions are evaluated at runtime and can contain references to variables, data sources, resources, functions, and other Terraform constructs. Interpolation enables you to dynamically generate values, compute derived values, and perform various transformations within your Terraform configurations.

The syntax for interpolation in Terraform is `${...}`. Inside the interpolation syntax, you can use a wide range of expressions, including variable references, resource attributes, function calls, and mathematical operations.

One of the errors encountered when applying Terraform configuration is the improper use of interpolation.

In this recipe, we'll illustrate an example of an interpolation error and how to resolve it.

Let's get started!

Getting ready

In this recipe, we have an existing Terraform configuration that provisions an Azure Resource Group that contains the following configuration:

```
variable "resource_group_name" {
  default = "rg-demo-error"
}

resource "azurerm_resource_group" "rg-app" {
  name = var.resource_group_name-test
  location = "westeurope"
}
```

In this configuration, the name of the Azure resource group is the value of the variable `resource_group_name`.

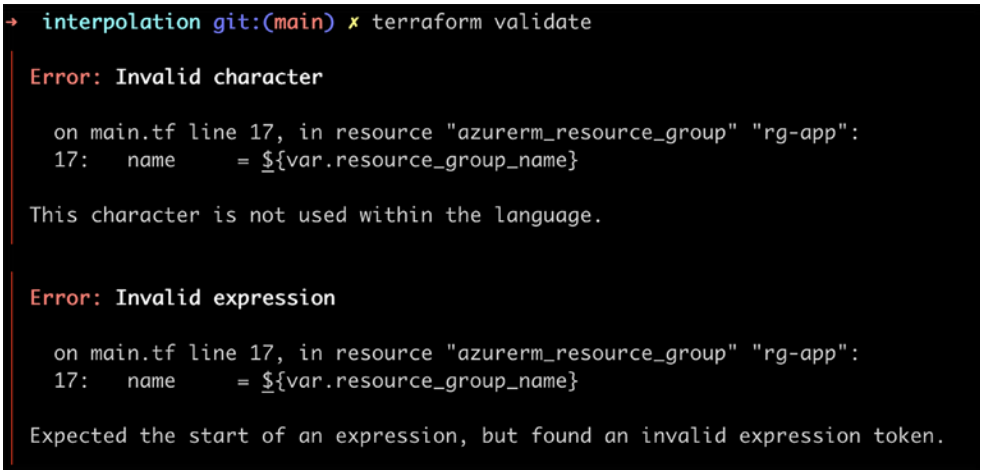
We will see in this recipe that this configuration contains an interpolation error and how to fix it.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP15/interpolation`.

How to do it...

Perform the following steps to apply this Terraform configuration:

1. In the folder that contains this configuration, run `terraform init`.
2. Then, run `terraform validate`.
3. The following screenshot shows the output of the `terraform validate` command:



```
→  interpolation git:(main) ✗ terraform validate

 Error: Invalid character

   on main.tf line 17, in resource "azurerm_resource_group" "rg-app":
   17:    name      = ${var.resource_group_name}

 This character is not used within the language.


 Error: Invalid expression

   on main.tf line 17, in resource "azurerm_resource_group" "rg-app":
   17:    name      = ${var.resource_group_name}

 Expected the start of an expression, but found an invalid expression token.
```

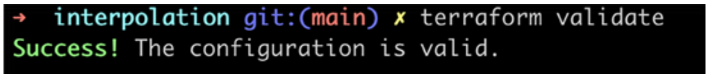
Figure 15.1: Terraform interpolation error

We can see that our configuration contains two errors in the resource group `name` attribute.

4. To fix these errors, we change the `name` value of the resource group to the following configuration:

```
resource "azurerm_resource_group" "rg-app" {
  name     = "${var.resource_group_name}"
  location = "westeurope"
}
```

5. Finally, we rerun the `terraform validate` command and we get the following output:



```
→  interpolation git:(main) ✗ terraform validate
Success! The configuration is valid.
```

Figure 15.2: Fixing Terraform interpolation

How it works...

In *Step 3*, we fixed the interpolation error by adding double quotes around the name value with the format `"${<variable>}<string>"`.

To check the validity of the configuration syntax, we use the `terraform validate` command, which we looked at in detail in *Chapter 6*, *Applying a Basic Terraform Workflow*, in the *Validating the code syntax* recipe.

To see more about `interpolation` troubleshooting, read the lab documentation here: `https://developer.hashicorp.com/terraform/tutorials/configuration-language/troubleshooting-workflow#correct-a-variable-interpolation-error`

See also

- Documentation for the `string` type and templating is available here: `https://developer.hashicorp.com/terraform/language/expressions/strings`

Fixing cycle errors

In *Chapter 2*, *Writing Terraform Configurations*, in the *Managing Terraform resource dependencies* recipe, we learned how to use implicit and explicit Terraform dependencies.

In some cases, when we use lots of dependencies between resources, we can get a cycle error between Terraform resources (such as cases where the resource depends on itself within the chain of dependencies).

To understand the concept of Terraform dependencies and cycle errors, read this explanation: `https://serverfault.com/a/1005791`

Let's get started!

Getting ready

For this recipe, we will start with the following configuration in `main.tf` (it's only a part of the configuration):

```
resource "azurerm_linux_virtual_machine" "vm" {
  name                                = "myvmdemo-${random_string.str.result}"
  network_interface_ids               = [azurerm_network_interface.nic.id]
  resource_group_name                 = azurerm_resource_group.rg.name
  location                            = azurerm_resource_group.rg.location

….

}

resource "azurerm_network_interface" "nic" {
  name                = "${azurerm_linux_virtual_machine.vm.name}-nic"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  ….
}
```

The complete source code of this Terraform configuration is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP15/cycle`.

The above configuration provisions an Azure VM and associates an Azure Network Interface (NIC), and we can see in this configuration that the VM has an implicit dependency on the NIC with the code `network_interface_ids = [azurerm_network_interface.nic.id]` and the NIC name contains the name of the VM with the configuration `name= "${azurerm_linux_virtual_machine.vm.name}-nic"`.

So, the VM depends on the NIC and the NIC depends on the VM.

In this recipe, we will see the issues produced by this configuration.

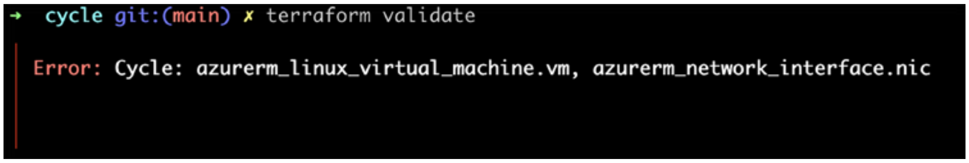
How to do it…

Perform the following steps to apply this configuration:

1. In the folder that contains this configuration, run the following commands:

   ```
   terraform init
   terraform validate
   ```

The following screenshot shows the output of these commands:



```
→  cycle git:(main) ✗ terraform validate

 Error: Cycle: azurerm_linux_virtual_machine.vm, azurerm_network_interface.nic
```

Figure 15.3: Terraform cycle error

We can see the cycle being reported between the VM and the NIC resources.

2. To fix this error, we can create a `locals` expression that contains the VM name, by adding the following Terraform configuration in the same `main.tf` file:

```
locals {
  vmname = "vmdemo-${random_string.str.result}"
}
Then, we use this locals expression in the name of the NIC with the
following Terraform configuration:
resource "azurerm_network_interface" "nic" {
  name                = "${local.vmname}-nic"
  resource_group_name = azurerm_resource_group.rg.name
…
```

3. Finally, we again run the `terraform validate` commands, which run successfully without a cycle error.

How it works…

In *Step 1*, we run the `terraform validate` command to check the syntax of our Terraform configuration. The `validate` command returns a cycle error. In *Step 2* of this recipe, we use a `locals` expression to remove a cycle dependency between the VM and the NIC.

There's more…

In this recipe, we use a locals expression to fix the cycle error. This isn't the only solution. To find the appropriate solution, analyze your Terraform configuration and find where the cycle dependency is.

To visualize all dependencies between the Terraform resources of your configuration, run the `terraform graph` command, and for more details, read the *Generating the graph dependencies* recipe of *Chapter 6*, *Applying a Basic Terraform Workflow* and the *Visualizing the Terraform resource dependencies with Rover* recipe in *Chapter 12*, *Deep-Diving into Terraform*.

To see more about cycle troubleshooting, read the lab documentation here: `https://developer.hashicorp.com/terraform/tutorials/configuration-language/troubleshooting-workflow#correct-a-cycle-error`

Fixing for_each errors

In *Chapter 3*, *Scaling Your Infrastructure with Terraform*, in the *Looping over object collections* recipe, we learned how to use the `for_each` expression to loop over collections.

Now, regarding `for_each` errors, potential issues related to `for_each` can include:

- **Missing Key in the Input Data Structure**: If the data structure used with `for_each` is empty or does not contain the expected keys or elements, Terraform may raise an error. For example, if you attempt to use `for_each` with an empty map or set, Terraform won't be able to create any instances, resulting in an error.
- **Duplicate Keys in the Input Data Structure**: `for_each` expects unique keys in the input data structure to create distinct resource instances. If there are duplicate keys, Terraform will raise an error because resource instances must have unique identifiers.
- **Data Type Mismatch**: The data structure used with `for_each` must be compatible with the resource or module block. If there's a type mismatch (for example, using a list instead of a map), Terraform will raise an error.
- **Referencing Nonexistent Items**: If you're using `for_each` to create instances based on a map, set, or other data structure, and you reference a non-existent key or element in your resource configuration, Terraform may raise an error during the planning or applying phase.

In this recipe, we will learn how to troubleshoot some data type mismatch errors that we can get when we use `for_each` expressions.

Let's get started!

Getting ready

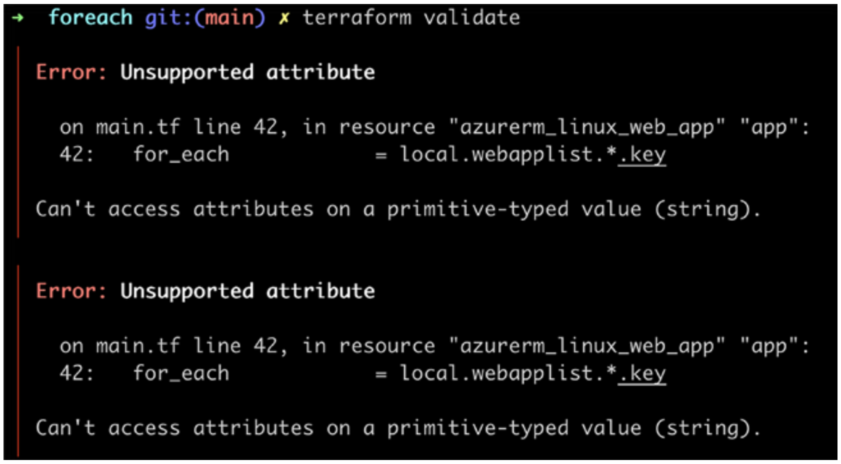
In this recipe, we will start with the following Terraform configuration:

```
locals {
  webapplist = ["webappdemobook1", "webappdemobook2"]
}

resource "azurerm_linux_web_app" "app" {
  for_each            = local.webapplist.*.key
  name                = "${each.value}-${random_string.str.result}"
  location            = "westeurope"
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id     = azurerm_service_plan.plan-app.id
  site_config {}
}
```

In this Terraform configuration, we want to create multiple Azure web apps that have different names and the same properties. To apply this Terraform configuration simply using one Terraform resource, we can use the `for_each` expression based on the local `webapplist` expression.

First, we run the `terraform init` command.

Then, the execution of `terraform validate` on the above configuration returns the following error:



```
→ foreach git:(main) ✗ terraform validate

Error: Unsupported attribute

  on main.tf line 42, in resource "azurerm_linux_web_app" "app":
  42:   for_each            = local.webapplist.*.key

Can't access attributes on a primitive-typed value (string).

Error: Unsupported attribute

  on main.tf line 42, in resource "azurerm_linux_web_app" "app":
  42:   for_each            = local.webapplist.*.key

Can't access attributes on a primitive-typed value (string).
```

Figure 15.4: Terraform for_each property error

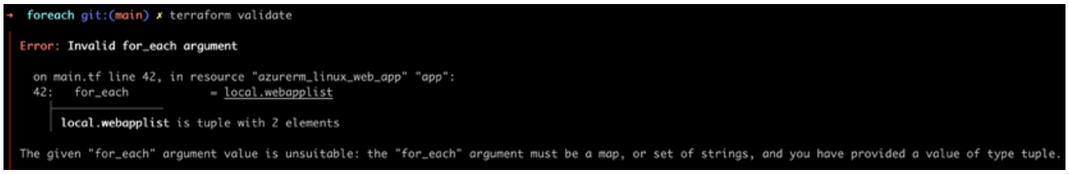
In this error, the key property of the `locals` expression is not recognized.

Another possibility for Terraform configuration with a `for_each` expression is the following:

```
resource "azurerm_linux_web_app" "app" {
  for_each            = local.webapplist
  name                = "${each.value}-${random_string.str.result}"
```

```
    location            = "westeurope"
    resource_group_name = azurerm_resource_group.rg-app.name
    service_plan_id     = azurerm_service_plan.plan-app.id
    site_config {}
  }
```

The execution of `terraform validate` on the above configuration returns the following error:



```
→ foreach git:(main) ✗ terraform validate

  Error: Invalid for_each argument

    on main.tf line 42, in resource "azurerm_linux_web_app" "app":
    42:   for_each            = local.webapplist
      ├────────────────
      │ local.webapplist is tuple with 2 elements

  The given "for_each" argument value is unsuitable: the "for_each" argument must be a map, or set of strings, and you have provided a value of type tuple.
```

Figure 15.05: Terraform for_each object error

In this error, the `for_each` expression must be applied on a map or set function. Here, we applied it on a tuple.

Let's see now how to fix the above two errors.

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP15/foreach`

How to do it…

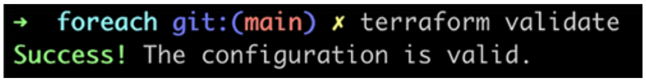
To fix the above two `for_each` errors, perform the following steps:

1. In the above configuration, update the configuration to the following:

```
resource "azurerm_linux_web_app" "app" {
  for_each            = toset(local.webapplist)
  name                = "${each.value}-${random_string.str.result}"
  location            = "westeurope"
  resource_group_name = azurerm_resource_group.rg-app.name
  service_plan_id     = azurerm_service_plan.plan-app.id
  site_config {}
}
```

In the above configuration, we fix the `for_each` expression to convert a `local` expression, `webapplist`, to a `set` of strings using the `toset` built-in Terraform function.

2. Then we run the `terraform validate` command on this configuration, and we get the following output:



```
→  foreach git:(main) ✗ terraform validate
Success! The configuration is valid.
```

Figure 15.6: Terraform fix on for_each

Now the configuration is valid, and we can run the basic Terraform workflow executing the `init`, `plan`, and `apply` commands.

There's more...

To address the potential errors mentioned in this recipe introduction, ensure that you have valid and correct data structures and that they align with the expected format for `for_each`. It's essential to verify the input data before using it with `for_each` to prevent errors during Terraform operations.

To read more about `for_each` troubleshooting, see the lab documentation here: `https://developer.hashicorp.com/terraform/tutorials/configuration-language/troubleshooting-workflow#correct-a-for_each-error`

Fixing output errors

In the previous recipe, we saw how to troubleshoot and fix `for_each` errors.

Now, using the same configuration, we can see another scenario to get the hostname for a created Azure web app. This will allow us to test the availability of the web app just after its creation. Note that when testing the avilabilty of the web app, we are simply working on the output render.

In this recipe, we will not discuss about the test of the availability of the Web App, we will just work on the output render.

Let's get started!

Getting ready

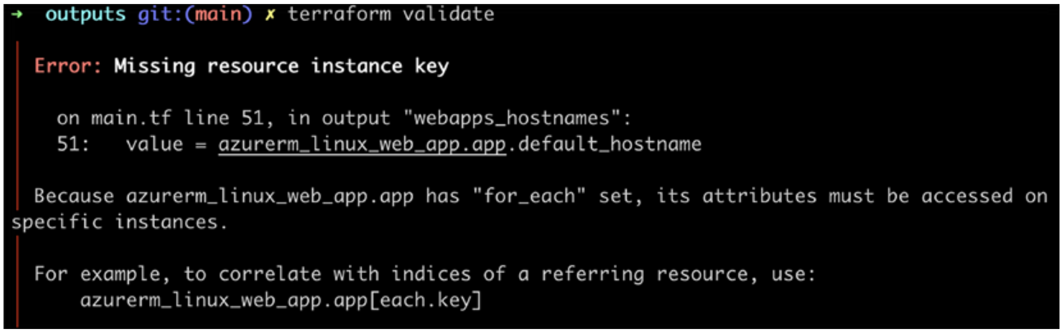
To add the `output`, in the same Terraform configuration as before, we add the following configuration:

```
output "webapps_hostnames" {
  value= azurerm_linux_web_app.app.default_hostname
}
```

The source code of this recipe is available here: `https://github.com/PacktPublishing/Terraform-Cookbook-Second-Edition/tree/main/CHAP15/outputs`

In this Terraform configuration, we run the `terraform init` command, and then the `terraform validate` command.

The following figure shows the result of the `validate` command:



```
→  outputs git:(main) ✗ terraform validate

│ Error: Missing resource instance key
│
│   on main.tf line 51, in output "webapps_hostnames":
│   51:    value = azurerm_linux_web_app.app.default_hostname
│
│ Because azurerm_linux_web_app.app has "for_each" set, its attributes must be accessed on
specific instances.
│
│ For example, to correlate with indices of a referring resource, use:
│     azurerm_linux_web_app.app[each.key]
```

Figure 15.7: Terraform output error

We get an error on `output` that says the `output` value isn't valid for resources provisioned with the `for_each` expression.

How to do it...

To fix this error, perform the following steps:

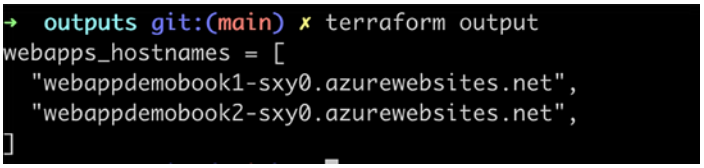
1. Update the above configuration to the following:

   ```
   output "webapps_hostnames" {
     value = [for app in azurerm_linux_web_app.app : app.default_
   hostname]
   }
   ```

   To fix the `output` error, we use the `for` expression in the `output` value.

   In the Looping over a map of object recipe of *Chapter 3*, *Scaling Your Infrastructure with Terraform*, we learned how to use `outputs` with loops on maps.

2. Then, run the Terraform workflow by executing the `terraform init`, `plan`, and `apply` commands.
3. Finally, run the `terraform outputs` command. The following screenshot shows the result of this command:



```
→  outputs git:(main) ✗ terraform output
webapps_hostnames = [
  "webappdemobook1-sxy0.azurewebsites.net",
  "webappdemobook2-sxy0.azurewebsites.net",
]
```

Figure 15.8: Fixing the output error

We can see the list of Azure web app hostnames in the Terraform `webapps_hostnames` `output` value.

How it works…

In *Step 1* we fixed the Terraform configuration on the existing output block by replacing `azurerm_linux_web_app.app.default_hostname` (which retrieves only one instance of `azurerm_linux_web_app`) with a `for` expression that loops over the list of created `azurerm_linux_web_app` instances. Then, in *Step 2* we executed the Terraform workflow to apply the configuration. Finally, in *Step 3*, we ran the `terraform output` command to display the `webapps_hostnames` output, so that we can see all the Web App hostnames.

There's more…

To read more about `output` troubleshooting, see the lab documentation here: `https://developer.hashicorp.com/terraform/tutorials/configuration-language/troubleshooting-workflow#correct-your-outputs-to-return-all-values`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/cloudanddevops`

Terraform Cheat Sheet

This Terraform cheat sheet is designed to provide quick answers and guidance to common tasks. Each section offers a concise overview of concepts, commands, and practices, making it easy to find the information you need. Whether you're creating cloud resources, managing infrastructure changes, or diving into advanced techniques, this cheat sheet is your companion in harnessing the power of Terraform for efficient and scalable infrastructure management.

Remember, while this cheat sheet covers a broad range of topics, Terraform is a dynamic and evolving tool. Always refer to the official documentation and community resources for the latest updates and insights into best practices.

This Terraform cheat sheet is not exhaustive and should be used in conjunction with the official Terraform documentation for accurate and up-to-date information.

The Terraform CLI documentation is available here: `https://developer.hashicorp.com/terraform/cli`

Basic commands

Command	What it does	Documentation link
`terraform -help`	Displays all CLI commands	`https://developer.hashicorp.com/terraform/cli/commands#basic-cli-features`
`terraform -version`	Displays the CLI version	`https://developer.hashicorp.com/terraform/cli/commands/version`
`terraform <command> -chdir`	Runs Terraform commands in a specific Terraform configuration folder	`https://developer.hashicorp.com/terraform/cli/commands#switching-working-directory-with-chdir`

Format Terraform configuration

Command	What it does	Documentation link
`terraform fmt`	Formats Terraform configuration	`https://developer.hashicorp.com/terraform/cli/commands/fmt`
`terraform fmt --recursive`	Formats Terraform configuration recursively	`https://developer.hashicorp.com/terraform/cli/commands/fmt#recursive`

Terraform providers management

Command	What it does	Documentation link
`terraform providers`	Returns the Terraform providers used in the current configuration	`https://developer.hashicorp.com/terraform/cli/commands/providers`
`terraform get -update=true`	Downloads Terraform providers	`https://developer.hashicorp.com/terraform/cli/commands/get`

Terraform dependency file

Command	What it does	Documentation link
`terraform init --upgrade`	Upgrades the Terraform dependency file	`https://developer.hashicorp.com/terraform/cli/commands/init#upgrade`
`terraform provider lock -platform=windows_amd64 -platform=linux_amd64`	Generates a Terraform lock for the specified OS	`https://developer.hashicorp.com/terraform/cli/commands/providers/lock`

Basic workflow commands

Command	What it does	Documentation link
`terraform init`	Initializes a directory of Terraform configuration files	`https://developer.hashicorp.com/terraform/cli/commands/init`
`terraform plan`	Visualizes a change preview	`https://developer.hashicorp.com/terraform/cli/commands/plan`
`terraform apply`	Applies changes	`https://developer.hashicorp.com/terraform/cli/commands/apply`

`terraform plan –out=out.tfplan`
`terraform apply out.tfplan`
	Visualizes changes and applies them using the plan file	`https://developer.hashicorp.com/terraform/cli/commands/plan#out-filename`

`terraform apply –auto-approve`	Auto-approves the application of changes	`https://developer.hashicorp.com/terraform/cli/commands/apply#auto-approve`
`terraform destroy`	Destroys all resources	`https://developer.hashicorp.com/terraform/cli/commands/destroy`
`terraform destroy –target <resource id>`	Destroys a specific resource	`https://developer.hashicorp.com/terraform/cli/commands/plan#target-address`

`terraform plan -out=tfplan`
`terraform show -json tfplan > plan.json`
	Exports the result of the plan in JSON format	

Backend configuration

Command	What it does	Documentation link
`terraform init –backend-config=backend.hcl`	Uses the backend configuration file	`https://developer.hashicorp.com/terraform/language/settings/backends/configuration#partial-configuration`

`terraform init -migrate-state`	Migrates the state file between backends	`https://developer.hashicorp.com/terraform/cli/commands/init#backend-initialization`

Validate configuration

Command	**What it does**	**Documentation link**
`terraform validate`	Validates Terraform configuration files	`https://developer.hashicorp.com/terraform/cli/commands/validate`

Get outputs

Command	**What it does**	**Documentation link**
`terraform output`	Views outputs	`https://developer.hashicorp.com/terraform/cli/commands/output`
`terraform output -json`	Views outputs in JSON format	`https://developer.hashicorp.com/terraform/cli/commands/output#json`

Import resources

Command	**What it does**	**Documentation link**
`terraform import <terraform resource id> <real resource id>`	Imports existing resources in state	`https://developer.hashicorp.com/terraform/cli/commands/import`

Terraform workspaces

Command	**What it does**	**Documentation link**
`terraform workspace select <name>`	Selects an existing workspace	`https://developer.hashicorp.com/terraform/cli/commands/workspace/select`
`terraform workspace select <name>  -or-create`	Selects an existing workspace or creates one if one doesn't already exist	`https://developer.hashicorp.com/terraform/cli/commands/workspace/select#or-create`
`terraform workspace new <name>`	Creates a new workspace	`https://developer.hashicorp.com/terraform/cli/commands/workspace/new`

`terraform workspace delete <name>`	Deletes a workspace	`https://developer.hashicorp.com/terraform/cli/commands/workspace/delete`
`terraform workspace list`	Lists all workspaces	`https://developer.hashicorp.com/terraform/cli/commands/workspace/list`
`terraform workspace show`	Displays the current workspace	`https://developer.hashicorp.com/terraform/cli/commands/workspace/show`

Terraform debug

Command	**What it does**	**Documentation link**
`terraform console`	Enters interactive console mode to evaluate Terraform expressions	`https://developer.hashicorp.com/terraform/cli/commands/console`

State management

Command	**What it does**	**Documentation link**
`terraform show`	Shows the state file	`https://developer.hashicorp.com/terraform/cli/commands/show`
`terraform state list`	Lists resources in the state	`https://developer.hashicorp.com/terraform/cli/commands/state/list`
`terraform state list <resource id>`	Lists specific resources in the state	`https://developer.hashicorp.com/terraform/cli/commands/state/list`
`terraform state rm`	Removes resources from the state	`https://developer.hashicorp.com/terraform/cli/commands/state/rm`
`terraform state mv`	Moves a resource from one reference name to another (within the same state)	`https://developer.hashicorp.com/terraform/cli/commands/state/mv`
`terraform state pull > <terraform.tfstate>`	Downloads state	`https://developer.hashicorp.com/terraform/cli/commands/state/pull`
`terraform state push`	Uploads local state to the backend	`https://developer.hashicorp.com/terraform/cli/commands/state/push`

`terraform state show`	Shows the resource in a state	`https://developer.hashicorp.com/terraform/cli/commands/state/show`
`terraform refresh (deprecated)`	Refreshes the state	`https://developer.hashicorp.com/terraform/cli/commands/refresh`
`terraform plan –refresh-only`	Previews the refresh of the state	`https://developer.hashicorp.com/terraform/cli/commands/refresh`
`terraform apply –refresh-only`	Applies the refresh of the state	`https://developer.hashicorp.com/terraform/cli/commands/refresh`
`terraform state replace-provider <provider source> <provider target>`	Replaces the provider in the state	`https://developer.hashicorp.com/terraform/cli/commands/state/replace-provider`

Display Terraform graph dependencies

Command	What it does	Documentation link
`terraform graph | dot -Tpng > graph.png`	Generates the Terraform graph dependency in a PNG file	`https://developer.hashicorp.com/terraform/cli/commands/graph`

Taint/untaint resources

Command	What it does	Documentation link
`terraform taint <resource id>`	Taints the resource	`https://developer.hashicorp.com/terraform/cli/commands/taint`
`terraform untaint <resource id>`	Untaints the resource	`https://developer.hashicorp.com/terraform/cli/commands/untaint`

Terraform Cloud/Enterprise

Command	What it does	Documentation link
`terraform login`	Logs in to Terraform Cloud/Enterprise	`https://developer.hashicorp.com/terraform/cli/commands/login`
`terraform logout`	Logs out of Terraform Cloud/Enterprise	`https://developer.hashicorp.com/terraform/cli/commands/logout`

Terraform Resources

This appendix contains a non-exhaustive list of official Terraform resources from HashiCorp and non-official resources from the community.

Terraform official resources

Documentation

- Terraform CLI documentation: `https://developer.hashicorp.com/terraform/cli`
- Terraform language documentation: `https://developer.hashicorp.com/terraform/language`
- Terraform Cloud: `https://developer.hashicorp.com/terraform/cloud-docs`
- Terraform Enterprise: `https://developer.hashicorp.com/terraform/enterprise`
- Cloud Development Kit for Terraform: `https://developer.hashicorp.com/terraform/cdktf`

Registry

- Terraform Registry: `https://registry.terraform.io/?product_intent=terraform`
- Terraform Registry publishing: `https://developer.hashicorp.com/terraform/registry`

Providers development

- Plugin development: `https://developer.hashicorp.com/terraform/plugin`
- Call APIs with custom framework providers: `https://developer.hashicorp.com/terraform/tutorials/providers-plugin-framework`
- Terraform Integration Program: `https://developer.hashicorp.com/terraform/docs/partnerships`
- Terraform glossary: `https://developer.hashicorp.com/terraform/docs/glossary`
- Terraform tools: `https://developer.hashicorp.com/terraform/docs/terraform-tools`
- Terraform tutorial library: `https://developer.hashicorp.com/tutorials/library?product=terraform`
- Terraform community forum: `https://discuss.hashicorp.com/c/terraform-core/27`

- Terraform support: `https://www.hashicorp.com/customer-success?product_intent=terraform`
- Terraform source code on GitHub: `https://github.com/hashicorp/terraform`
- Terraform releases: `https://github.com/hashicorp/terraform/releases`
- HashiCorp YouTube channel: `https://www.youtube.com/@HashiCorp`

Terraform community resources

- YouTube channel of Ned Bellavance: `https://www.youtube.com/@NedintheCloud`
- Awesome Terraform resources: `https://github.com/shuaibiyy/awesome-terraform`
- Terraform on Azure documentation: `https://learn.microsoft.com/en-us/azure/developer/terraform/`
- Terraform on GCP documentation: `https://cloud.google.com/docs/terraform`
- Terraform articles on Medium: `https://medium.com/search?q=terraform`
- Udemy courses on Terraform: `https://www.udemy.com/topic/terraform/`

Terraform news feed

- Weekly Terraform: `https://www.weekly.tf/`

Terraform certifications and certification preparation

Terraform certification program pages

- `https://developer.hashicorp.com/certifications/infrastructure-automation`
- `https://developer.hashicorp.com/terraform/tutorials/certification-003`

Terraform certification preparation

- `https://www.pluralsight.com/paths/hashicorp-certified-terraform-associate`
- `https://www.udemy.com/course/terraform-hands-on-labs/`
- `https://www.udemy.com/course/terraform-associate-practice-exam/`
- `https://www.pluralsight.com/cloud-guru/courses/hashicorp-certified-terraform-associate`

`packt.com`

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

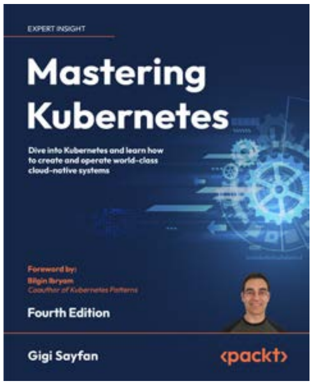
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At `www.packt.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Mastering Kubernetes - Fourth Edition

Gigi Sayfan

ISBN: 9781804611395

- Learn how to govern Kubernetes using policy engines
- Learn what it takes to run Kubernetes in production and at scale
- Build and run stateful applications and complex microservices
- Master Kubernetes networking with services, Ingress objects, load balancers, and service meshes
- Achieve high availability for your Kubernetes clusters
- Improve Kubernetes observability with tools such as Prometheus, Grafana, and Jaeger
- Extend Kubernetes with the Kubernetes API, plugins, and webhooks

AWS for Solutions Architects - Second Edition

Saurabh Shrivastava , Neelanjali Srivastav, Alberto Artasanchez, Imtiaz Sayed

ISBN: 9781803238951

- Optimize your Cloud Workload using the AWS Well-Architected Framework
- Learn methods to migrate your workload using the AWS Cloud Adoption Framework
- Apply cloud automation at various layers of application workload to increase efficiency
- Build a landing zone in AWS and hybrid cloud setups with deep networking techniques
- Select reference architectures for business scenarios, like data lakes, containers, and serverless apps
- Apply emerging technologies in your architecture, including AI/ML, IoT and blockchain

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Terraform Cookbook, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review` page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

Abstract Syntax Tree (AST) 373

Advanced Package Tool (APT) package manager 13
  used, for installing Terraform on Linux 13, 14

alias
  adding, to provider to use multiple instances of same provider 40-42

Ansible
  reference link 410

App Service instance
  creating, on App Service instance 105

apply command 165

archive_file resource
  reference link 114

ArgoCD tools 350

ARM templates
  executing, in Terraform 256-260
  reference link 256

automation mode
  Terraform, running in 446-448

AWS
  S3 backend, using 307-311
  Terraform, using for 302-307

AzApi Terraform provider
  using 294-299

Azure CLI commands
  executing, in Terraform 261-264

Azure Cloud Shell
  Terraform, using in 242-245

Azure credential provider
  protecting 246-250

Azure Key Vault
  using, with Terraform to protect secrets 265-271

Azure remote backend
  state file, protecting 251-256

Azure Repos (Azure DevOps) 230

AzureRM Provider
  reference link 132

Azure serverless infrastructure
  building, with Terraform 277- 279

azure-specific data source 146

Azure VM
  configuring, with Terraform 271-277
  provisioning, with Terraform 271- 277

B

BDD terraform-compliance
  reference link 381

Behavior-Driven Development (BDD) 377

built-in jsonencode function
  reference link 59

built-in yamldecode function
  reference link 59

C

check block
  reference link 72

checkov rules 354

checks

using, for infrastructure validation 70-72

Chocolatey 11, 227

reference link, for commands 13

URL 11

used, for installing Terraform on Windows 11, 12

CI/CD pipelines

building, for Terraform configurations in Azure Pipelines 451-460

building, for Terraform modules in Azure Pipelines 469- 475

workspaces, working with 465- 469

Cloud Development Kit for Terraform (CDKTF)

benefits 435

reference link 435

using 437

code syntax

validating 162-164

Command-Line Interface (CLI) 34

conditional expressions

reference link 61

writing 59, 61

console command 195

reference link 196

continuous integration (CI) pipeline 164

count meta-argument

multiple resources, provisioning with 81-85

reference link 85

create_before_destroy property 417

custom functions

local variables, used for 49-51

custom pre- and post-conditions

adding 66-70

references 70

Custom Resource Deployment (CRD) 339

cycle errors

fixing 556- 558

cycle troubleshooting

reference link 559

D

data blocks 106

reference link 107

data sources

external data, obtaining with 104-106

dependency file 31

dependency graph

generating 188, 189

dependency lock file

reference link 32

Desired State Configuration (DSC) 393

destroy command 168

executing 403

reference link 168

Docker

references 20

Docker container

creating, with Terraform 331-334

Terraform, executing in 17-20

Docker Engine

reference link 331

Dockerfiles

reference link 17

Docker installation guides

reference link 17

Don't Repeat Yourself (DRY) principle 2, 75

dynamic block

multiple blocks, generating with 93- 97

E

environments
managing, with workspaces 177-179

environment variables
reference link 198

existing resources
importing 137- 143

explicit dependency 65

external data
obtaining, with data sources 104- 106
querying, with Terraform 107-111

external resources
using, from Terraform State files 143-145

external Terraform resource
references 111

F

feature flags 85

Flux CLI
installation link 345

fmt command 159-161
reference link 162

for_each errors
fixing 559-562

for_each troubleshooting
reference link 562

G

GCP
GCS backend, using 319-323
Terraform, using for 312- 319

GCP Cloud Shell
Terraform, executing 323- 327

GCS backend
using, in GCP 319-323

Git hooks
used, for checking configuration before committing code 429-432

GitHub Actions
Terraform execution, automating 460-465
Terraform module publishing workflow, building with 476- 481

Go
URL 382

graph command 189
reference link 190

Graphviz
URL 189, 190

H

HashiCorp 209

HashiCorp Cloud Platform (HCP)
reference link 483

HashiCorp Configuration Language (HCL) 157, 257

hcl2json
URL 373

HCL enhancements
reference link 36

Helm Chart
deploying, in Kubernetes with Terrafrom 340-343

Helm CLI
reference link 340

helm_release documentation
reference link 343

I

Identity and Access Management(IAM) 302

implicit dependency 65

import block
reference link 140

import command
reference link 143

Infracost
Azure cost of infrastructure, estimating with 288- 294

Infracost run task
integrating, during Terraform Cloud run 540-547

Infrastructure as Code (IaC) 1, 75, 157, 158, 402
best practices 2

infrastructure provisioning
in multiple environments 76

infrastructure resources
destroying 166, 167

infrastructure validation
checks, using for 70-72

init
reference link 27

Inspec 389, 392
URL 389

Inspec profiles
reference link 392

installation of Terraform, with package manager
reference link 14

integrated cost estimation
using, for cloud resources 535-540

integrated Terraform module integration test
using 395- 399

interpolation 554

interpolation errors
fixing 554-556

interpolation troubleshooting
reference link 556

J

jq 15, 165, 182
reference link 15, 185

JSON
output, exporting in 181-184

K

Kitchen-supported provisioners
reference link 393

Kitchen-supported verifiers
reference link 394

Kitchen-Terraform 354, 388
references 394
used, for testing Terraform configuration 388-394

kitchen test command
reference link 394

kreuzwerker/docker
reference link 334

Kubernetes
Terraform, used for deploying Helm Chart 340-343

Kubernetes controller
using, as Terraform reconciliation loop 344-349

Kubernetes resources
deploying, with Terraform 334-339

L

Linux
APT package manager, used for installing Terraform on 13, 14
script, used for installing Terraform on 14-16

local backend
reference link 127

local-exec provisioner 116
reference link 117

local_file resource
reference link 114

local files
manipulating, with Terraform 111-113

local programs
executing, with Terraform 114, 115

local Terraform State
using 124-127

local variables
reference link 51
using, for custom functions 49-51

loop functionalities
using 89-92

M

Man in The Middle (MITM) attacks 31

manual changes
allowing 408, 409
ignoring 406, 407-409

map of objects
looping 89-92

maps
filtering 97-100
using 85-89

merge function
reference link 89

moved configuration block
reference link 154

multiple blocks
generating, with dynamic block 93-97

multiple resources
provisioning, with count meta-argument 81-85

multiple Terraform versions
switching between 21-25

N

Network Security Group (NSG) rule 415

O

OPA binary 367

OPA policies
testing 372

OPA query 368

Open Policy Agent (OPA) 354, 366, 372
features 367
reference link 367
Terraform configuration compliance, checking with 526- 535
using, to check Terraform configuration 369-372

Open Policy Agent (OPA), usage artifacts
data input 368
OPA binary 367
OPA query 368
Rego policies 368

output errors
fixing 562, 564

outputs
exporting, in JSON 181-184
used, for exposing Terraform provisioned data 52- 54

output command
reference link 185

output troubleshooting
reference link 564

P

Packer
reference link 277, 417

passwords
generating, with Terraform 61-63

Pester 355, 356
pros and cons 361, 362
references 362
used, for applying compliance 356-361

Pester test file structure
reference link 360

Pester tests
writing 357, 358

plan command 96

prevent_destroy property 404, 406

private Git repository
used, for sharing Terraform module 229-234

private module registry
Terraform Cloud, using as 510-515

provider alias
reference link 42

providers, properties
reference link 40

provider lock command
reference link 174

provider version
using 34-39

providers command 170
reference link 171

public registry
Terraform module, sharing with GitHub 213-217
Terraform module, using from 209-212

PXF certificate 427

pytest 362
reference link 363
references 366
Terraform configuration, testing 363-366

Python
Terraform compliance, testing with 362-366

R

refresh command
reference link 137

Rego playground
URL 372

Rego policies 368

remote backend
using, in Terraform Cloud 500-506

remote operations 516

resource already exists error
reference link 553

resources
preventing, from getting destroyed 402-404
tainting 185, 186

RG-APP-IMPORT
reference link 138

Rover
Terraform resource dependencies, visualizing with 433

Rspec 389

Ruby
references 388

S

S3 backend
- using, for AWS 307-311

S3 remote backend
- reference link 311

Save action plugin
- reference link 162

script
- used, for installing Terraform on Linux 14-16

self-signed SSL certificate
- generating, with Terraform 425-427

sensitive variables
- references 49
- securing 47-49

Shell scripts
- executing, with Terraform Shell provider 117-120

show command
- reference link 129

state command
- reference link 188

state mv command
- cons 153
- pros 153
- reference link 154

state list command
- reference link 130

state rm command
- reference link 131

state show command
- reference link 131

string type and templating, documentation
- reference link 556

T

taint operation
- applying 185
- reference link 188

taint command
- reference link 188

template 221

templating feature
- using 410-413

Terrafile 235

Terrafile pattern
- applying, for using Terraform module 235-239

Terraform 157
- ARM templates, executing 256-260
- authenticating, to Terraform Cloud 486-490
- Azure CLI commands, executing 261-264
- best practices 2, 3
- configuring 34-39
- downloading 4
- executing, in Docker container 17-20
- executing, in GCP Cloud Shell 323-327
- execution, automating in GitHub Actions 460-465
- external data, querying with 107-111
- installing on Linux, APT package manager used 13, 14
- installing on Linux, script used 14-16
- installing on Windows, Chocolatey used 11, 12
- local files, manipulating 111-113
- local programs, executing with 114, 115
- manual installation, on Windows 4-10
- provider version, using 34-39
- resources dependencies, managing 64-66
- running, in automation mode 446-448

templating feature, using 410- 414
used, for creating Docker container 331-334
used, for deploying Kubernetes resources 334-339
used, for downloading azurerm provider 39
used, for generating passwords 61-63
used, for generating self-signed SSL certificate 425-427
used, for generating self-signed SSL certificate 427
used, for zero-downtime deployment 415-417
using, for AWS 302-307
using, for GCP 312-319
using, in Azure Cloud Shell 242-245
using, to deploy Helm Chart in Kubernetes 340-343

Terraform azurerm provider
reference link 36

Terraform binary versioning
reference link 40

Terraform block, properties
reference link 40

Terraform built-in functions
calling 54-56
reference link 56

Terraform CDK
using, for developers 435-443

Terraform CLI 157
URL 515

Terraform Cloud (TFC) 483
configuring, with Terraform TFE provider 547-551
reference link 501
remote backend, using in 500-506
Terraform, authenticating to 486-490
Terraform configuration, remote execution 516-525
Terraform State, migrating to 506-510
URL 515
using, as private module registry 510-515
workspaces, managing 490-499

Terraform Cloud run
Infracost run task, executing 540-547

terraform-compliance 354, 377
analyzing, with tfsec 375, 376
references 382
testing, with Python 362-366
used, for applying Terraform compliance 378-381

Terraform configuration 157
best practices 2, 3
checking, with OPA 369-372
contact tests 354
generating, for existing Azure infrastructure 280-287
integration tests 354
keeping clean 158-160
list of providers, displaying 169, 170
plan analysis for compliance 354
resources, refactoring 147-154
sharing 175
testing, with Kitchen-Terraform 388-394
testing, with pytest 363-366
unit tests 353
YAML files, using in 57, 59

Terraform configuration compliance
checking, with OPA 526-535

Terraform configuration dependencies
managing, with Terragrunt 418- 421

Terraform configuration directories
using 190-192

Terraform configuration folder hierarchy
implementing 76- 80

Terraform configurations, in Azure Pipelines
CI/CD pipelines, building for 451-460

Terraform debug
reference link 198

Terraform dependencies and cycle errors
reference link 556

Terraform dependency file
reference link 174

Terraformer 281

Terraform execution
debugging 196, 197

Terraform expression
evaluating 193-195
testing 193-195

Terraform lock file
generating, with Windows and Linux compatibility 171-174

Terraform module 200
creating 201- 203
documentation, generating 225-228
file, using 218-220
multiple instances, provisioning 206, 207
output, with loop 209
sharing, in public registry with GitHub 213-217
sharing, with private Git repository 229-234
structure 205
structure, generating from 222-224
using, for applying Terrafile pattern 235- 239
using, from public registry 209, 211, 212
using, it locally 204-206

Terraform module code
testing, with Terratest 383-387

Terraform module configuration
copying 175, 176

Terraform module generator
using 221-224

Terraform module publishing workflow
building, with GitHub Actions 476-481

Terraform modules, in Azure Pipelines
CI/CD pipelines, building for 469-475

Terraform moved block
cons 153
pros 153
reference link 154

Terraform Official Packaging guide
reference link 14

Terraform Operator 350

Terraform outputs
reference link 54

terraform plan execution summary
displaying 449, 450, 451

Terraform plugin
reference link 162

Terraform providers
reference link 32
upgrading 27- 31

Terraform provisioned data
exposing, with outputs 52-54

Terraform reconciliation loop
Kubernetes controller, using as 344- 349

terraform_remote_state block
reference link 146

terraform_remote_state data source 146

Terraform resource dependencies
visualizing, with Rover 433, 434

Terraform, resources dependencies
reference link 66

Terraform Shell provider
reference link 121
used, for executing Shell scripts 117-120

Terraform State 157
content, displaying 128, 129
detailed resource properties, displaying 130
migrating, to Terraform Cloud 506- 510
reference link 123
resource names, listing 130
resources, deleting 131
resources, managing 127, 128
synchronizing 132-137

Terraform State errors
reference link 553

Terraform State files
external resources, using 143-145

Terraform State management
reference link 132

Terraform testing
reference link 400

Terraform TFE provider
Terraform Cloud, configuring with 547- 551

Terraform troubleshooting
reference link 553

Terraform variables
refernce link 46

Terragrunt 20
reference link 418
Terraform configuration dependencies, managing with 418- 420
using, as wrapper for Terraform 422-424

Terratest 354, 382, 387, 388
references 388
Terraform module code, testing 383-387

Test-Driven Design (TDD) 396

test command
reference link 400

test pyramid
reference link 353

TFC workspace
creating 499, 500

tfenv 21
commands 26
functionalities 21
installing 22
reference link 22
references 27

tflint 354

tfsec 354, 373
benefits 374
reference link 373
references 376, 377
used, for analyzing Terraform configuration 375, 376

tfvc 170
reference link 171

U

untaint command
reference link 188

untaint operation
reference link 188

V

validate command 164
reference link 166

variable custom rules validation
refernce link 46

variables
elements 44
manipulating 43, 44, 45, 46

virtual network (VNet) 271

W

Windows
  Chocolatey, used for installing Terraform on 11, 12
  Terraform, installing manually on 4-10
workspace command
  reference link 181
workspaces 177, 180, 181
  managing, in Terraform Cloud 490-499
  reference link 181
  used, for managing environments 177-179
workspaces, CI/CD
  working with 465-469

Y

YAML files
  using, in Terraform configuration 57-59

Z

zero-downtime deployment
  with Terraform 415- 418
zipmap function
  reference link 413

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781804616420`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly