Clean Code Principles and Patterns

A Software Practitioner's Handbook

PETRI SILÉN

ISBN: 9798373835732

Table of Contents

About the Author

Petri Silén is a seasoned software developer working at Nokia Networks in Finland with industry experience of almost 30 years.He has done both frontend and backend development with a solid competence in multiple programming languages, including C++, Java, and JavaScript/TypeScript. He started his career at Nokia Telecommunications in 1995. During his first years, he developed a real-time mobile networks analytics product called "Traffica" in C++ for major telecom customers worldwide, including companies like T-Mobile, Orange, Vodafone, and Claro. The initial product was for monitoring a 2G circuit-switched core network and GPRS packet-switched core network. Later, functionality to Traffica was added to cover new network technologies, like 3G circuit-switched and packet core networks, 3G radio networks, and 4G/LTE. He later developed new functionality for Traffica using Java and web technologies, including jQuery and React. During the last few years, he has developed cloud-native containerized microservices with Java and C++ for the next-generation Customer and Networks Insights (CNI) product used by major communications service providers like Verizon, AT&T, USCC, and KDDI. The main application areas he has contributed during the last years include KPI-based real-time alerting, anomaly detection for KPIs, and configurable real-time data exporting.

During his free time, he has developed a data visualization application using React, Redux, TypeScript, and Jakarta EE. He has also developed a security-first cloud-native microservice framework for Node.js in TypeScript. He likes to take care of his Kaapo cat, take walks, play tennis and badminton, ski in the winter, and watch soccer and ice hockey on TV.

Introduction

This book teaches you how to write clean code. It presents software design and development principles and patterns in a very practical manner. This book is suitable for both junior and senior developers. Basic understanding and knowledge of object-oriented programming in one language, like C++, Java, JavaScript/TypeScript, Python or C# is required. Examples in this book are presented in Java, JavaScript/TypeScript, or C++. Most examples are in Java or JavaScript/TypeScript and are adaptable to other programming languages, too. The content of this book is divided into eleven chapters.

The *second chapter* is about *architectural design principles* that enable the development of true cloud-native microservices. The first architectural design principle described is the single responsibility principle which defines that a piece of software should be responsible for one thing at its abstraction level. Then a uniform naming principle for microservices, clients, APIs, and libraries is presented. The encapsulation principle defines how each software component should hide the internal state behind a public API. The service aggregation principle is introduced with a detailed explanation of how a higher-level microservice can aggregate lower-level microservices. Architectural patterns, like event sourcing, command query responsibility segregation (CQRS), and distributed transactions, are discussed. Distributed transactions are covered with examples using both the saga orchestration pattern and the saga choreography pattern. You get answers on how to avoid code duplication at the architectural level. Externalized configuration principle describes how service configuration should be handled in modern environments. We discuss the service substitution principle, which states that dependent services a microservice uses should be easily substitutable. The importance of autopilot microservices is discussed from statelessness, resiliency, high availability, observability, and automatic scaling point of view. Towards the end of the chapter, there is a discussion about different ways microservices can communicate with each other. Several rules are presented on how to version software components. And the chapter ends with a discussion of why it is helpful to limit the number of technologies used in a software system.

The *third chapter* presents *object-oriented design principles*. We start the chapter by describing all the SOLID principles: Single responsibility principle, open-closed principle, Liskov's substitution

principle, interface segregation principle, and dependency inversion principle. Each SOLID principle is presented with realistic but simple examples. The uniform naming principle defines a uniform way to name interfaces, classes, functions, function pairs, boolean functions (predicates), builder, factory, conversion, and lifecycle methods. The encapsulation principle describes that a class should encapsulate its internal state and how immutability helps ensure state encapsulation. The encapsulation principle also discusses the importance of not leaking an object's internal state out. The object composition principle defines that composition should be preferred over inheritance. Domain-driven design (DDD) is presented with two real-world examples. All the design patterns from *GoF's Design Patterns* book are presented with realistic yet simple examples. The don't ask, tell principle is presented as a way to avoid the feature envy design smell. The chapter also discusses avoiding primitive-type obsession and the benefits of using semantically validated function arguments. The chapter ends by presenting the dependency injection principle and avoiding code duplication principle, also known as the don't repeat yourself (DRY) principle

The *fourth chapter* is about *coding principles*. The chapter starts with a principle for uniformly naming variables in code. A uniform naming convention is presented for integer, floating-point, boolean, string, enum, and collection variables. Also, a naming convention is defined for maps, pairs, tuples, objects, optionals, and callback functions. The uniform source code repository structure principle is presented with examples for C++, Java, and JavaScript/TypeScript. Next, the avoid comments principle defines concrete ways to remove unnecessary comments from the code. The following concrete actions are presented: naming things correctly, returning a named value, return-type aliasing, extracting a constant for a boolean expression, extracting a constant for a complex expression, extracting enumerated values, and extracting a function. The chapter discusses the benefits of using a statically typed language. We discuss the most common refactoring techniques: renaming, extracting a method, extracting a variable, replacing conditionals with polymorphism, and introducing a parameter object. The importance of static code analysis is described, and the most popular static code analysis tools for C++, Java, and JavaScript/TypeScript are listed. The most common static code analysis issues are listed with the preferred way to correct them. Handling errors and exceptions correctly in code is fundamental and can be easily forgotten or done wrong. This chapter instructs how to handle errors and exceptions, how to handle Java's checked exceptions, and how to return errors by returning a boolean failure indicator, an optional value, or an error object. The chapter instructs how to adapt code to a wanted error-handling mechanism, handle errors in asynchronous code, handle stream errors, and handle errors functionally. The null value handling is discussed. The ways to avoid off-by-one errors are presented. Readers are instructed on handling situations where some code is copied from a web page found by googling. The chapter ends with a discussion about code optimization: when and how to optimize.

The *fifth chapter* is dedicated to *testing principles*. We start with the introduction of the functional testing pyramid. Then we present unit testing and instruct how to use test-driven development (TDD). We give unit test examples with mocking in Java, JavaScript, and C++. When introducing software component integration testing, we discuss behavior-driven development (BDD) and the Gherkin language to describe features. Integration test examples are given using Cucumber for Java

and Postman API development platform. The chapter also discusses the integration testing of UI software components. We end the integration testing section with an example of setting up an integration testing environment using Docker Compose. Lastly, the purpose of end-to-end (E2E) testing is discussed with some examples. The chapter ends with a discussion about non-functional testing. The following categories of non-functional testing are covered in more detail: performance testing, stability testing, reliability testing, security testing, stress, and scalability testing.

The *sixth chapter* handles *security principles*. The threat modeling process is introduced. A full-blown OpenID Connect/OAuth 2.0 authentication and authorization example with TypeScript, Vue.js, and Keycloak is implemented. Then we discuss how authorization by validating a JWT should be handled in the backend. Examples are presented with Node.js, Express, Java, and Spring Boot. The chapter ends with a discussion of the most important security features: password policy, cryptography, denial-of-service prevention, SQL injection prevention, security configuration, automatic vulnerability scanning, integrity, error handling, audit logging, and input validation.

The *seventh chapter* is about *API design principles*. First, we tackle design principles for frontend facing APIs. We discuss how to design JSON-RPC, REST, and GraphQL APIs. Also, subscription-based and real-time APIs are presented with realistic examples using Server-Sent Events (SSE) and the WebSocket protocol. The last part of the chapter discusses inter-microservice API design and event-driven architecture. gRPC is introduced as a synchronous inter-microservice communication method, and examples of request-only and request-response asynchronous APIs are presented.

The *8th chapter* discusses *databases and related principles*. We cover the following types of databases: relational databases, document databases (MongoDB), key-value databases (Redis), wide-column databases (Cassandra), and search engines. For relational databases, we present how to use object-relational mapping (ORM), one-to-one, one-to-many and many-to-many relationships, and parameterized SQL queries. Finally, we present three normalization rules for relational databases.

The *9th chapter* presents *concurrent programming principles* regarding threading, parallel algorithms, and thread safety. For thread safety, we present several ways to achieve thread synchronization: synchronization directives, atomic variables, mutexes, and spinlocks.

The *10th chapter* discusses *teamwork principles*. We explain the importance of using an agile framework and discuss the fact that a developer usually never works alone and what that entails. We discuss how to document a software component so that onboarding new developers is easy and quick. Technical debt in software is something that each team should avoid. Some concrete actions to prevent technical debt are presented. Code reviews are something teams should do, and this chapter gives guidance on what to focus on in code reviews. The chapter ends with a discussion of developer roles each team should have and provides hints on enabling a team to develop software as concurrently as possible.

The *11th chapter* is dedicated to *DevSecOps*. DevOps describes practices that integrate software development (Dev) and software operations (Ops). It aims to shorten the software development life

cycle through parallelization and automation and provides continuous delivery with high software quality. DevSecOps is a DevOps augmentation where security practices are integrated into the DevOps practices. This chapter presents the phases of the DevOps lifecycle: plan, code, build and test, release, deploy, operate and monitor. The chapter gives an example of creating a microservice container image and how to specify the deployment of a microservice to a Kubernetes cluster. Also, a complete example of a CI/CD pipeline using GitHub Actions is provided.

Architectural Principles

This chapter describes architectural principles for designing clean, modern cloud-native software systems and applications. Cloud-native software is built of loosely coupled scalable, resilient and observable services that can run in public, private, or hybrid clouds. Cloud-native software utilizes technologies like containers (e.g., Docker), microservices, serverless functions, and container orchestration (e.g., Kubernetes), and it can be automatically deployed using declarative code.

This chapter discusses the following architectural principles and patterns:

- Single responsibility principle
- Uniform naming principle
- Encapsulation principle
- Service aggregation principle
- High cohesion, low coupling principle
- Library composition principle
- Avoid duplication principle
- Externalized service configuration principle
- Service substitution principle
- Autopilot microservices principle
    - Stateless microservices principle
    - Resilient microservices principle
    - Horizontally autoscaling microservices principle
    - Highly-available microservices principle
    - Observable services principle
- Inter-service communication patterns
- Domain-driven architectural design principle
- Software versioning principles
- Git Version control principle

- Architectural patterns
- Preferred technology stacks principle

Software Hierarchy

A *software system* consists of multiple computer programs and anything related to those programs to make them operable, including but not limited to configuration, deployment code, and documentation. A software system is divided into two parts: *the backend* and *the frontend*. Backend software runs on servers, and frontend software runs on client devices like PCs, tablets, and phones. Backend software consists of *services*. Frontend software consist of *clients* that use backend services and *standalone applications* that do not use any backend services. An example of a standalone application is a calculator or a simple text editor.

The term *application* is often used to describe a single program designated for a specific purpose. In general, a software application is some software applied to solve a specific problem. From an end user's point of view, all clients are applications. But from a developer's point of view, an application needs both a client and backend service(s) to be functional unless the application is a *standalone*

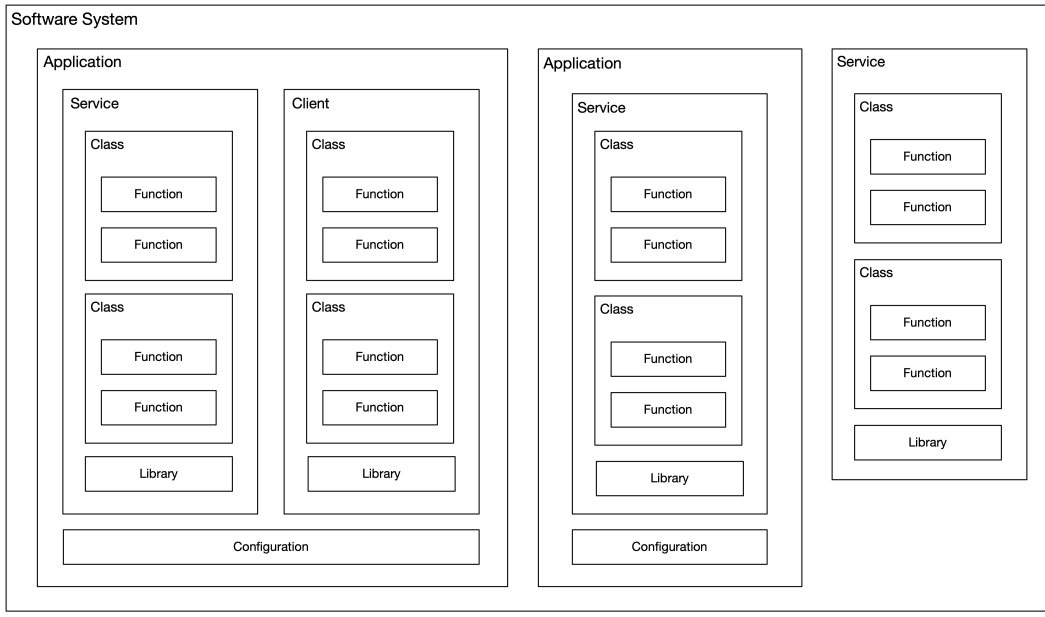

Fig 2.1 Software Hierarchy

application. In this book, I will use the term application to designate a logical grouping of program(s) and related artifacts, like configuration, to form a functional piece of the software

system dedicated to a specific purpose. In my definition, a non-standalone application consists of one or more services and possibly a client or clients to fulfill an end user's need. Let's say we have a software system for telecom network analytics. That system provides data visualization functionality. We can call the data visualization part of the software system a data visualization application. That application consists of, for example, a web client and two services, one for fetching data and one for configuration. Suppose we also have a generic data ingester microservice in the system. That generic data ingester is not an application without some configuration that makes it a specific service that we can call an application. For example, the generic data ingester can have a configuration to ingest raw data from a radio network. The generic data ingester and the configuration together form an application: a radio network data ingester.

Computer programs and *libraries* are *software components*. A *software component* is something that can be individually packaged, tested, and delivered. It consists of one or more classes, and a class consists of one or more functions (class methods). (There are no traditional classes in purely functional languages, but software components consist only of functions.) A computer program can also be composed of one or more libraries, and a library can be composed of other libraries.

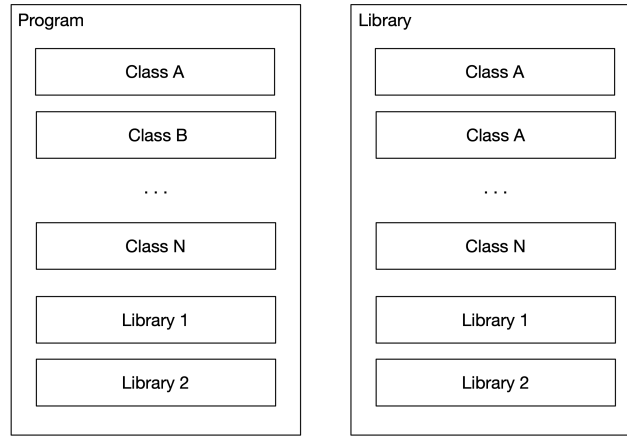

Fig 2.2 Software Components

Single Responsibility Principle

> *A software entity should have only single responsibility at its abstraction level.*

A software system is at the highest level in the software hierarchy and should have a single dedicated purpose. For example, there can be an e-commerce or payroll software system. But there should not be a software system that handles both e-commerce and payroll-related activities. If you were a software vendor and had made an e-commerce software system, selling that to clients wanting an e-commerce solution would be easy. But if you had made a software system that

encompasses both e-commerce and payroll functionality, it would be hard to sell that to customers wanting only an e-commerce solution because they might already have a payroll software system and, of course, don't want another one.

Let's consider the application level in the software hierarchy. Suppose we have designed a software system for telecom network analytics. This software system is divided into four different applications: Radio network data ingestion, core network data ingestion, data aggregation, and data visualization. Each of these applications has a single dedicated purpose. Suppose we had coupled the data aggregation and visualization applications into a single application. In that case, replacing the data visualization part with a 3rd party application could be difficult. But when they are separate applications with a well-defined interface, it would be much easier to replace the data visualization application with a 3rd party application, if needed.

A software component should also have a single dedicated purpose. A service type of software component with a single responsibility is called a *microservice*. For example, one microservice could be responsible for handling orders and another for handling sales items. Both of those microservices are responsible for one thing only. We should not have a microservice responsible for both orders and sales items. That would be against the single responsibility principle because order and sales item handling are two different functionalities at the same level of abstraction.

There are many advantages to microservices:

- Improved productivity
- You can choose the best-suited programming language and technology stack
- Microservices are easy to develop in parallel because there will be fewer merge conflicts
- Developing a monolith can result in more frequent merge conflicts
- Improved resiliency and fault isolation
    - A fault in a single microservice does not bring other microservices down
    - A bug in a monolith can bring the whole monolith down
- Better scalability
    - Stateless microservices can be automatically horizontally scalable
    - Horizontal scaling of a monolith is complicated or impossible
- Better data security and compliance
    - Each microservice encapsulates its data, which can be accessed via a public API only
- Faster and easier upgrades
    - Upgrading only the changed microservice(s) is enough. No need to update the whole monolith every time
- Faster release cycle
    - Build the changed microservice only. No need to build the whole monolith when something changes
- Fewer dependencies

    - Lower probability for dependency conflicts
- Enables *open-closed architecture*, meaning architecture that is open for extension and closed for modification
    - New functionality not related to any existing microservice can be put into a new microservice instead of modifying the current codebase.

The main drawback of microservices is the complexity that a distributed architecture brings. Operating and monitoring a microservice-based software system is complicated. Also, testing a distributed system is more challenging than testing a monolith. Development teams should put focus on these areas by hiring DevOps and test automation specialists.

A library type of software component should also have a single responsibility. Like calling single-responsibility services microservices, we can call a single-responsibility library a *microlibrary*. For example, there could be a library for handling YAML-format content and another for handling XML-format content. We shouldn't try to bundle the handling of both formats into a single library. If we did and needed only the YAML-related functionality, we would also always get the XML-related functionality. Our code would always ship with the XML-related code, even if it is never used. This can introduce unnecessary code bloat. We would also have to take any security patch for the library into use, even if the patch was only for the XML-related functionality we don't use.

Uniform Naming Principle

> *Name microservices with a **service** or **api** postfix, clients with a **client** postfix, and libraries with a **library** postfix.*

When developing software, you should establish a naming convention for microservices, clients, and libraries. The preferred naming convention for microservices is *<service's purpose>-service*. For example: *data-aggregation-service* or *email-sending-service*. Use the microservice name systematically in different places. For example, use it as the Kubernetes Deployment name and the source code repository name (or directory name in case of a monorepo). It is enough to name your microservices with the *service* postfix instead of a *microservice* postfix because each service should be a microservice by default. So, there would not be any real benefit in naming microservices with the microservice postfix. That would just make the microservice name longer without any added value.

If you want to be more specific in naming microservices, you can name API microservices with an *api* postfix instead of the more generic *service* postfix, for example, *sales-item-api*. In this book, I am not using the *api* postfix but always use the *service* postfix only.

The preferred naming convention for clients is *<client's purpose>-<client type>-client*. For

example: *data-visualization-web-client*, *data-visualization-mobile-client*, *data-visualization-android-client* or *data-visualization-ios-client*.

The preferred naming convention for libraries is *<library's purpose>-library*. For example: *common-utils-library* or *common-ui-components-library*.

When using these naming conventions, a clear distinction between a microservice, client, and library-type software component can be made only by looking at the name. Also, it is easy to recognize if a source code repository contains a microservice, client, or library.

Encapsulation Principle

> *Microservice must encapsulate its internal state behind a public API. Anything behind the public API is considered private to the microservice and cannot be accessed directly by other microservices.*

Microservices should define a public API that other microservices use for interfacing. Anything behind the public API is private and inaccessible from other microservices.

While microservices should be made stateless (the s*tateless services principle* is discussed later in this chapter), a stateless microservice needs a place to store its state outside the microservice. Typically, the state is stored in a database. The database is the microservice's internal dependency and should be made private to the microservice, meaning that no other microservice can directly access the database. Access to the database happens indirectly using the microservice's public API.

It is discouraged to allow multiple microservices to share a single database because then there is no control how each microservice will use the database, and what requirements each microservice has for the database.

Sometimes it is possible to share a *physical* database with several microservices if each microservice uses its own *logical* database. This requires that a specific database user is created for each microservice. Each database user can access only one logical database dedicated to a particular microservice. In this way, no microservice can directly access another microservice's database. This approach can still pose some problems because the dimensioning requirements of all microservices for the shared physical database must be considered. Also, the deployment responsibility of the shared database must be decided. The shared database could be deployed as a platform or common service as part of the platform or common services deployment, for example.

Service Aggregation Principle

Service on a higher level of abstraction aggregates services on a lower level of abstraction.

Service aggregation happens when one service on a higher level of abstraction aggregates services on a lower level of abstraction.

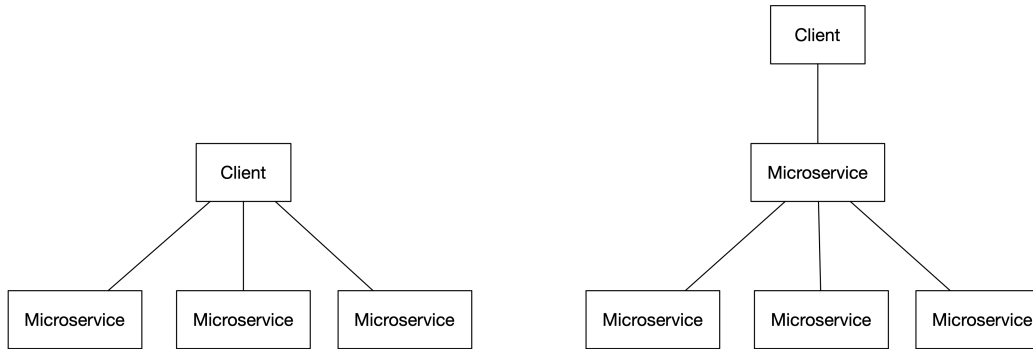

Fig 2.3 Architecture Without and With Service Aggregation

Let's have a service aggregation example with an e-commerce software system that allows people to sell second-hand products online.

The problem domain of the e-commerce service consists of the following subdomains:

- User account domain
    - Create, modify, and delete a user account
    - View user account with sales items and orders
- Sales item domain
    - Add new sales items, modify, view, and delete sales items
- Shopping cart domain
    - Add/remove sales items to/from a shopping cart, empty a shopping cart
    - View shopping cart with sales item details
- Order domain
    - Placing orders
        - Ensure payment
        - Create order
        - Remove ordered items from the shopping cart
        - Mark ordered sales items sold

        - Send order confirmation by email
    - View orders with sales item details
    - Update and delete orders

We should not implement all the subdomains in a single *ecommerce-service* microservice because then we would not be following the *single responsibility principle*. We should use service aggregation. We create a separate lower-level microservice for each subdomain. Then we create a higher-level *ecommerce-service* microservice that aggregates those lower-level microservices.

We can define that our *ecommerce-service* aggregates the following lower-level microservices:

- *user-account-service*
    - Create/Read/Update/Delete user accounts
- *sales-item-service*
    - Create/Read/Update/Delete sales items
- *shopping-cart-service*
    - View a shopping cart, add/remove sales items from a shopping cart or empty a shopping cart
- *order-service*
    - Create/Read/Update/Delete orders
- *email-notification-service*
    - Send email notifications

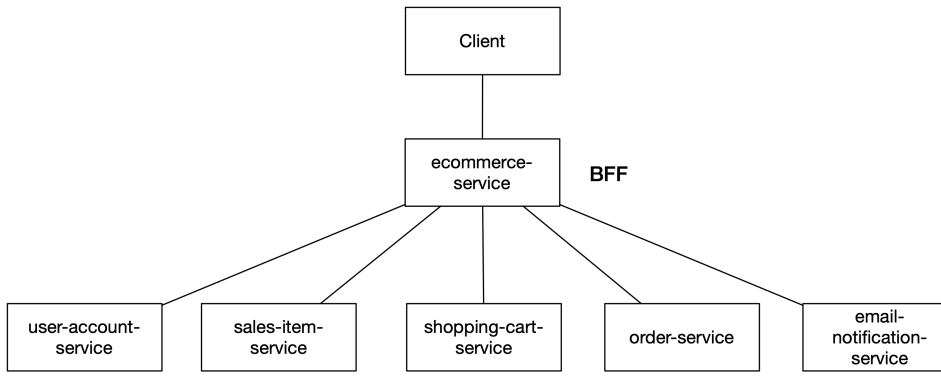

Fig 2.4 Service Aggregation in E-Commerce Software System

Most of the microservices described above can be implemented as REST APIs because they mainly contain basic CRUD (create, read, update and delete) operations for which a REST API is a good match. We will handle API design in more detail in a later chapter. Let's implement the sales-item-

service as a REST API using Java and Spring Boot. We will implement the `SalesItemController` class first. It defines API endpoints for creating, getting, updating, and deleting sales items:

SalesItemController.java

```
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.tags.Tag;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(SalesItemController.API_ENDPOINT)
@Tag(
  name = "Sales item API",
  description = "Manages sales items"
)
public class SalesItemController {
  public static final String API_ENDPOINT = "/sales-items";

  @Autowired
  private SalesItemService salesItemService;

  @PostMapping
  @ResponseStatus(HttpStatus.CREATED)
  @Operation(summary = "Creates new sales item")
  public final SalesItem createSalesItem(
    @RequestBody final SalesItemArg salesItemArg
  ) {
    return salesItemService.createSalesItem(salesItemArg);
  }

  @GetMapping
  @ResponseStatus(HttpStatus.OK)
  @Operation(summary = "Gets sales items")
  public final Iterable<SalesItem> getSalesItems() {
    return salesItemService.getSalesItems();
  }

  @GetMapping("/{id}")
  @ResponseStatus(HttpStatus.OK)
  @Operation(summary = "Gets sales item by id")
  public final SalesItem getSalesItemById(
    @PathVariable("id") final Long id
  ) {
    return salesItemService.getSalesItemById(id);
  }

  @GetMapping(params = "userAccountId")
  @ResponseStatus(HttpStatus.OK)
  @Operation(summary = "Gets sales items by user account id")
  public final Iterable<SalesItem> getSalesItemsByUserAccountId(
```

```
    @RequestParam("userAccountId") final Long userAccountId
  ) {
    return salesItemService
              .getSalesItemsByUserAccountId(userAccountId);
  }

  @PutMapping("/{id}")
  @ResponseStatus(HttpStatus.NO_CONTENT)
  @Operation(summary = "Updates a sales item")
  public final void updateSalesItem(
    @PathVariable final Long id,
    @RequestBody final SalesItemArg salesItemArg
  ) {
    salesItemService.updateSalesItem(id, salesItemArg);
  }

  @DeleteMapping("/{id}")
  @ResponseStatus(HttpStatus.NO_CONTENT)
  @Operation(summary = "Deletes a sales item by id")
  public final void deleteSalesItemById(
    @PathVariable final Long id
  ) {
    salesItemService.deleteSalesItemById(id);
  }

  @DeleteMapping
  @ResponseStatus(HttpStatus.NO_CONTENT)
  @Operation(summary = "Deletes all sales items")
  public final void deleteSalesItems() {
    salesItemService.deleteSalesItems();
  }
}
```

As we can notice from the above code, the `SalesItemController` class delegates the actual work to an instance of a class that implements the `SalesItemService` interface. This is an example of using the *bridge pattern* which is discussed, along with other design patterns, in the next chapter. In the bridge pattern, the controller is just an abstraction of the service, and a class implementing the `SalesItemService` interface provides a concrete implementation. We can change the service implementation without changing the controller or introduce a different controller, e.g., a GraphQL controller, using the same `SalesItemService` interface. Only by changing the used controller class could we change the API from a REST API to a GraphQL API. Below is the definition of the `SalesItemService` interface:

SalesItemService.java

```
public interface SalesItemService {
  SalesItem createSalesItem(SalesItemArg salesItemArg);
  SalesItem getSalesItemById(Long id);

  Iterable<SalesItem> getSalesItemsByUserAccountId(
   Long userAccountId
  );

  Iterable<SalesItem> getSalesItems();
  void updateSalesItem(Long id, SalesItemArg salesItemArg);
  void deleteSalesItemById(Long id);
  void deleteSalesItems();
}
```

The below `SalesItemServiceImpl` class implements the `SalesItemService` interface. It will interact with a sales item repository to persist, fetch and delete data to/from a database.

SalesItemServiceImpl.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class SalesItemServiceImpl implements SalesItemService {
  private static final String SALES_ITEM = "Sales item";

  @Autowired
  private SalesItemRepository salesItemRepository;

  @Override
  public final SalesItem createSalesItem(
    final SalesItemArg salesItemArg
  ) {
    final var salesItem = SalesItem.from(salesItemArg);
    return salesItemRepository.save(salesItem);
  }

  @Override
  public final SalesItem getSalesItemById(final Long id) {
    return salesItemRepository.findById(id)
              .orElseThrow(() ->
                new EntityNotFoundError(SALES_ITEM, id));
  }

  @Override
  public final Iterable<SalesItem> getSalesItemsByUserAccountId(
    final Long userAccountId
  ) {
    return salesItemRepository
              .findByUserAccountId(userAccountId);
  }

  @Override
  public final Iterable<SalesItem> getSalesItems() {
    return salesItemRepository.findAll();
  }

  @Override
  public final void updateSalesItem(
    final Long id,
    final SalesItemArg salesItemArg
  ) {
    if (salesItemRepository.existsById(id)) {
      final var salesItem =
        SalesItem.from(salesItemArg, id);

      salesItemRepository.save(salesItem);
    } else {
      throw new EntityNotFoundError(SALES_ITEM, id);
    }
  }

  @Override
  public final void deleteSalesItemById(final Long id) {
    if (salesItemRepository.existsById(id)) {
      salesItemRepository.deleteById(id);
```

```
    }
  }

  @Override
  public final void deleteSalesItems() {
    salesItemRepository.deleteAll();
  }
}
```

EntityNotFoundError.java

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class EntityNotFoundError extends RuntimeException {
  EntityNotFoundError(final String entityType, final long id) {
    super(entityType +
          " entity not found with id " +
          String.valueOf(id));
  }
}
```

The `SalesItemRepository` interface is defined below. Spring will create an instance of a class implementing that interface and inject it into an instance of the `SalesItemServiceImpl` class. The `SalesItemRepository` interface extends Spring's `CrudRepository` interface, which provides many database access methods by default. It provides the following and more methods: `findAll`, `findById`, `save`, `existsById`, `deleteAll`, and `deleteById`. We need to add only one method to the `SalesItemRepository` interface: `findByUserAccountId`. Spring will automatically generate an implementation for the `findByUserAccountId` method because the method name follows certain conventions of the Spring Data (https://docs.spring.io/spring-data/jpa/docs/current/reference/html) framework. We just need to add the method to the interface, and that's it. We don't have to provide an implementation for the method because Spring will do it for us.

SalesItemRepository.java

```
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface SalesItemRepository extends
                    CrudRepository<SalesItem, Long>
{
  Iterable<SalesItem> findByUserAccountId(Long userAccountId);
}
```

Next, we define the `SalesItem` entity class, which contains properties like `name` and `price`. It also includes two methods to convert an instance of the `SalesItemArg` Data Transfer Object (DTO) class to an instance of the `SalesItem` class. A DTO is an object that transfers data between a server and a client. I have used the class name `SalesItemArg` instead of `SalesItemDto` to describe that a `SalesItemArg` DTO is an argument for an API endpoint. If some API endpoint

returned a special sales item DTO instead of a sales item entity, I would name that DTO class `SalesItemResponse` instead of `SalesItemDto`. The terms *Arg* and *Response* better describe the direction in which a DTO transfers data. You could also use the following DTO names: `InputSalesItem` and `OutputSalesItem` to describe an incoming and outgoing DTO (from the server's point of view).

SalesItem.java

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.modelmapper.ModelMapper;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class SalesItem {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private Long userAccountId;

  @NotNull
  private String name;

  @Min(value = 0, message = "Price must be greater than 0")
  @Max(
    value = Integer.MAX_VALUE,
    message = "Price must be <= " + Integer.MAX_VALUE
  )
  private Integer price;

  static SalesItem from(final SalesItemArg salesItemArg) {
    return new ModelMapper()
                .map(salesItemArg, SalesItem.class);
  }

  static SalesItem from(
    final SalesItemArg salesItemArg,
    final Long id
  ) {
    final var salesItem =
      new ModelMapper().map(salesItemArg, SalesItem.class);

    salesItem.setId(id);
    return salesItem;
  }
}
```

The below `SalesItemArg` class contains the same properties as the `SalesItem` entity class, except the `id` property. The `SalesItemArg` DTO class is used when creating a new sales item or updating an existing sales item. When creating a new sales item, the `id` property should not be given by the client because the microservice will automatically generate it (or the database will, actually, in this case).

SalesItemArg.java

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class SalesItemArg {
  private Long userAccountId;
  private String name;
  private Integer price;
}
```

Below is defined how the *ecommerce-service* will orchestrate the use of the aggregated lower-level microservices:

- User account domain
    - Delegates CRUD operations to *user-account-service*
    - Delegates to *sales-item-service* to fetch information about user's sales items
    - Delegates to *order-service* to fetch information about user's orders
- Sales item domain
    - Delegates CRUD operations to *sales-item-service*
- Shopping cart domain
    - Delegates read/add/remove/empty operations to *shopping-cart-service*
    - Delegates to *sales-item-service* to fetch information about the sales items in the shopping cart
- Order domain
    - Ensures that payment is confirmed by the payment gateway
    - Delegates CRUD operations to *order-service*
    - Delegates to *shopping-cart-service* to remove bought items from the shopping cart
    - Delegates to *sales-item-service* for marking sales items bought
    - Delegates to *email-notification-service* for sending order confirmation email
    - Delegates to *sales-item-service* to fetch information about order's sales items

The *ecommerce-service* is meant to be used by frontend clients, like a web client, for example. *Backend for Frontend* (BFF) term is often used to describe a microservice designed to provide an API for frontend clients. Compared to the BFF term, service aggregation is a generic term, and

there need not be a frontend involved. You can use service aggregation to create an aggregated microservice used by another microservice or microservices. There can even be multiple levels of service aggregation if you have a large and complex software system.

Clients can have different needs regarding what information they want from an API. For example, a mobile client might be limited to exposing only a subset of all information available from an API. In contrast, a web client can fetch all information, or it can be customized what information a client retrieves from the API.

All of the above requirements are something that a GraphQL-based API can fulfill. For that reason, it would be wise to implement the *ecommerce-service* using GraphQL. I have chosen JavaScript, Node.js, and Express as technologies to implement a single GraphQL query in the *ecommerce-service*. Below is the implementation of a `user` query, which fetches data from three microservices. It fetches user account information from the *user-account-service*, the user's sales items from the *sales-item-service*, and finally, the user's orders from the *order-service*.

server.js

```
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { buildSchema, GraphQLError } = require('graphql');
const axios = require('axios').default;

const schema = buildSchema(`
  type UserAccount {
    id: ID!,
    userName: String!
    # Define additional properties...
  }

  type SalesItem {
    id: ID!,
    name: String!
    # Define additional properties...
  }

  type Order {
    id: ID!,
    userId: ID!
    # Define additional properties...
  }

  type User {
    userAccount: UserAccount!
    salesItems: [SalesItem!]!
    orders: [Order!]!
  }

  type Query {
    user(id: ID!): User!
  }
`);

const {
  ORDER_SERVICE_URL,
  SALES_ITEM_SERVICE_URL,
  USER_ACCOUNT_SERVICE_URL
```

```
} = process.env;

const rootValue = {
  user: async ({ id }) => {
    try {
      const [
        { data: userAccount },
        { data: salesItems },
        { data: orders }
      ] = await Promise.all([
        axios.get(`${USER_ACCOUNT_SERVICE_URL}/user-accounts/${id}`),
        axios.get(
          `${SALES_ITEM_SERVICE_URL}/sales-items?userAccountId=${id}`
        ),
        axios.get(`${ORDER_SERVICE_URL}/orders?userAccountId=${id}`)
      ]);

      return {
        userAccount,
        salesItems,
        orders
      };
    } catch (error) {
      throw new GraphQLError(error.message);
    }
  },
};

const app = express();

app.use('/graphql', graphqlHTTP({
  schema,
  rootValue,
  graphiql: true,
}));

app.listen(4000);
```

After you have started the above program with the `node server.js` command, you can access the GraphiQL endpoint with a browser at *http://localhost:4000/graphql*

On the left-hand side pane, you can specify a GraphQL query. For example, to query the user identified with id 2:

```
{
  user(id: 2) {
    userAccount {
      id
      userName
    }
    salesItems {
      id
      name
    }
    orders {
      id
      userId
    }
  }
}
```

Because we haven't implemented the lower-level microservices, let's modify the part of the *server.js* where lower level microservices are accessed to return dummy static results instead of accessing the real lower-level microservices:

```
const [
        { data: userAccount },
        { data: salesItems },
        { data: orders }
      ] = await Promise.all([
        Promise.resolve({
          data: {
            id,
            userName: 'pksilen'
          }
        }),
        Promise.resolve({
          data: [
            {
              id: 1,
              name: 'sales item 1'
            }
          ]
        }),
        Promise.resolve({
          data: [
            {
              id: 1,
              userId: id
            }
          ]
        })
      ]);
```

If we now execute the previously specified query, we should see the following query result:

```
{
  "data": {
    "user": {
      "userAccount": {
        "id": "2",
        "userName": "pksilen"
      },
      "salesItems": [
        {
          "id": "1",
          "name": "sales item 1"
        }
      ],
      "orders": [
        {
          "id": "1",
          "userId": "2"
        }
      ]
    }
  }
}
```

We can simulate a failure by modifying the *server.js* to contain the following code:

```
const [
        { data: userAccount },
        { data: salesItems },
        { data: orders }
      ] = await Promise.all([
        axios.get(`http://localhost:3000/user-accounts/${id}`),
        Promise.resolve({
          data: [
            {
              id: 1,
              name: 'sales item 1'
            }
          ]
        }),
        Promise.resolve({
          data: [
            {
              id: 1,
              userId: id
            }
          ]
        })
      ]);
```

Now, if we execute the query again, we will get the below error response because the server cannot connect to a service at the local host on port 3000 because there is no service running at *localhost:3000*.

```
{
  "errors": [
    {
      "message": "connect ECONNREFUSED 127.0.0.1:3000",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "user"
      ],
      "extensions": {}
    }
  ],
  "data": null
}
```

You can also query a user and specify the query to return only a subset of fields. The below query does not return ids and does not return orders. The server-side GraphQL library automatically includes only requested fields in the response. You, as a developer, do not have to do anything. You can, of course, optimize your microservice to fetch only the requested fields from the database if you desire.

```
{
  user(id: 2) {
    userAccount {
      userName
```

```
    }
    salesItems {
      name
    }
  }
}
```

The result for the above query will be the following:

```
{
  "data": {
    "user": {
      "userAccount": {
        "userName": "pksilen"
      },
      "salesItems": [
        {
          "name": "sales item 1"
        }
      ]
    }
  }
}
```

The above example lacks some features like authorization that is needed for production. Authorization should check that a user can only execute the `user` query to fetch his/hers resources. The authorization should fail if a user tries to execute the *user* query using someone else's id. Security is discussed more in the coming *security principles* chapter.

The `user` query in the previous example spanned over multiple lower-level microservices: *user-account-service*, *sales-item-service*, and *order-service*. Because the query is not mutating anything, it can be executed without a distributed transaction. A distributed transaction is similar to a regular (database) transaction, with the difference that it spans multiple remote services.

The API endpoint for placing an order in the *ecommerce-service* needs to create a new order using the *order-service*, mark purchased sales items as bought using the *sales-item-service*, empty the shopping cart using the *shopping-cart-service*, and finally send order confirmation email using the *email-notification-service*. These actions need to be wrapped inside a distributed transaction because we want to be able to roll back the transaction if any of these operations fail. Guidance on how to implement a distributed transaction is given later in this chapter.

Service aggregation utilizes the *facade pattern*. The facade pattern allows hiding individual lower-level microservices behind a facade (the higher-level microservice). The clients of the software system access the system through the facade. They don't directly contact the individual lower-level microservices behind the facade because it breaks the encapsulation of the lower-level microservices inside the higher-level microservice. A client accessing the lower-level microservices directly creates unwanted coupling between the client and the lower-level microservices, which makes changing the lower-level microservices hard without affecting the client.

Think about a post office counter as an example of a real-world facade. It serves as a facade for the

post office and when you need to receive a package, you communicate with that facade (the post office clerk at the counter). You have a simple interface of just telling the package code, and the clerk will find the package from the correct shelf and bring it to you. If you hadn't that facade, it would mean that you would have to do lower-level work by yourself. Instead of just telling the package code, you must walk to the shelves and try to find the proper shelf where your package is located, make sure that you pick the correct package, and then carry the package by yourself. In addition to requiring more work, this approach is more error-prone. You can accidentally pick someone else's package if you are not pedantic enough. And think about the case when you go to the post office next time and find out that all the shelves have been rearranged. This wouldn't be a problem if you used the facade.

Service aggregation, where a higher-level microservice delegates to lower-level microservices, also implements the *bridge pattern*. A higher-level microservice provides only some high-level control and relies on the lower-level microservices to do the actual work.

Service aggregation allows using more *design patterns* from the object-oriented design world. The most useful design patterns in the context of service aggregation are:

- Decorator pattern
- Proxy pattern
- Adapter pattern

Decorator pattern can be used to add functionality in a higher-level microservice for lower-level microservices. One example is adding audit logging in a higher-level microservice. For example, you can add audit logging to be performed for requests in the *ecommerce-service*. You don't need to implement the audit logging separately in all the lower-level microservices.

Proxy pattern can be used to control the access from a higher-level microservice to lower-level microservices. Typical examples of the proxy pattern are authorization and caching. For example, you can add authorization and caching to be performed for requests in the *ecommerce-service*. Only after successful authorization will the requests be delivered to the lower-level microservices. And if a request's response is not found in the cache, the request will be forwarded to the appropriate lower-level microservice. You don’t need to implement authorization and caching separately in all the lower-level microservices.

Adapter pattern allows a higher-level microservice to adapt to different versions of the lower-level microservices while maintaining the API towards clients unchanged.

High Cohesion, Low Coupling Principle

> *A software system should consist of services with high cohesion and low coupling.*

Cohesion refers to the degree to which classes inside a service belong together. Coupling refers to how many other services a service is interacting with.

When following the *single responsibility principle*, it is possible to implement services as microservices with high cohesion. Service aggregation adds low coupling. Microservices and service aggregation together enable high cohesion and low coupling, which is the target of good architecture. If there were no service aggregation, lower-level microservices would need to communicate with each other, creating high coupling in the architecture. Also, clients would be coupled with the lower-level microservices. For example, in the e-commerce example, the *order-service* would be coupled with almost all the other microservices. And if the *sales-item-service* API changed, in the worst case, there would be a change needed in three other microservices. When using service aggregation, lower-level microservices are coupled only to the higher-level microservice.

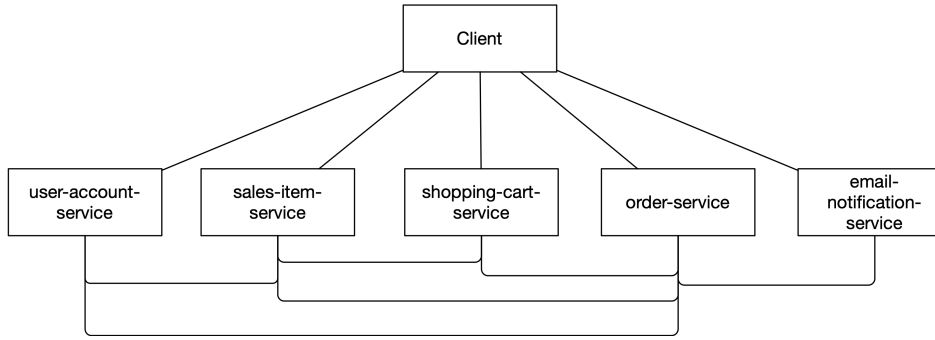

Fig 2.5 E-Commerce Software System With High Coupling

High cohesion and low coupling mean that the development of services can be highly parallelized. In the e-commerce example, the five lower-level microservices don't have coupling with each other. The development of each of those microservices can be isolated and assigned to a single team member or a group of team members. The development of the lower-level microservices can proceed in parallel, and the development of the higher-level microservice can start when the APIs of the lower-level microservices become stable enough. The target is to design the lower-level microservices APIs early on to enable the development of the higher-level microservice.

Library Composition Principle

> *Higher-level libraries should be composed of lower-level libraries.*

Suppose you need a library for parsing configuration files (in particular syntax) in YAML or JSON

format. In that case, you can first create the needed YAML and JSON parsing libraries (or use existing ones). Then you can create the configuration file parsing library, composed of the YAML and JSON parsing libraries. You would then have three different libraries: one higher-level library and two lower-level libraries. Each library has a single responsibility: one for parsing JSON, one for parsing YAML, and one for parsing configuration files with a specific syntax, either in JSON or YAML. Software components can now use the higher-level library for parsing configuration files, and they need not be aware of the JSON/YAML parsing libraries at all.

Avoid Duplication Principle

> *Avoid software duplication at the software system and service level.*

Duplication at the software system level happens when two or more software systems use the same services. For example, two different software systems can both have a message broker, API gateway, identity and access management (IAM) application, and log and metrics collection services. You could continue this list even further. The goal of duplication-free architecture is to have only one deployment of these services. Public cloud providers offer these services for your use. If you have a Kubernetes cluster, an alternative solution is to deploy your software systems in different Kubernetes namespaces and deploy the common services to a shared Kubernetes namespace, which can be called the *platform* or *common-services*, for example.

Duplication at the service level happens when two or more services have common functionality that could be extracted to a separate new microservice. For example, consider a case where both a *user-account-service* and *order-service* have the functionality to send notification messages by email to a user. This email-sending functionality is duplicated in both microservices. Duplication can be avoided by extracting the email-sending functionality to a separate new microservice. The single responsibility of the microservices becomes more evident when the email-sending functionality is extracted to its own microservice. One might think another alternative is extracting the common functionality to a library. This is not a solution that is as good because microservices become dependent on the library. When changes to the library are needed (e.g., security updates), you must change the library version in all the microservices using the library and then test all the affected microservices.

When a company develops multiple software systems in several departments, the software development typically happens in silos. The departments are not necessarily aware of what the other departments are doing. For example, it might be possible that two departments have both developed a microservice for sending emails. There is now software duplication that none is aware of. This is not an optimal situation. A software development company should do something to enable collaboration between the departments and break the silos. One good way to share software is to establish shared folders or organizations in the source code repository hosting service that the company uses. For example, in GitHub, you could create an organization for sharing source code

repositories for common libraries and another for sharing common services. Each software development department has access to those common organizations and can still develop its software inside its own GitHub organization. In this way, the company can enforce proper access control for the source code of different departments, if needed. When a team needs to develop something new, it can first consult the common source code repositories to find out if something is already available that can be reused as such or extended.

Externalized Service Configuration Principle

Service configuration means any data that varies between service deployments (different environments, different customers, etc.).

> *The configuration of a service should be externalized. It should be stored in the environment where the service is running, not in the source code. Externalized configuration makes the service adaptable to different environments and needs.*

The following are typical places where externalized configuration can be stored when software is running in a Kubernetes cluster:

- Environment variables
- Kubernetes ConfigMaps
- Kubernetes Secrets

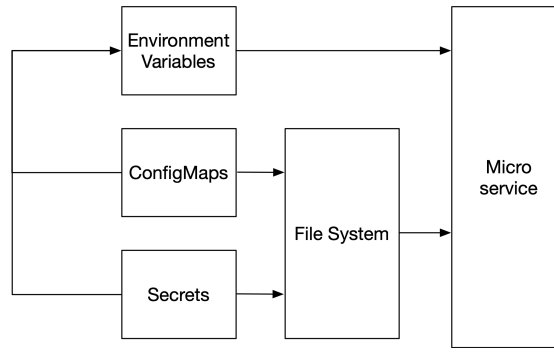

Fig 2.6 Configuration Storage Options

In the following sections, let's discuss these three configuration storage options.

Environment Variables

Environment variables can be used to store configuration as simple key-value pairs. They are typically used to store information like how to connect to dependent services, like a database or a message broker, or a microservice's logging level. Environment variables are available for the running process of a microservice, which can access the environment variable values by their names (keys).

You should not hardcode the default values for environment variables in the source code. This is because the default values are typically not for a production environment but for a development environment. Suppose you deploy a service to a production environment and forget to set all the needed environment variables. In that case, your service will have some environment variables with default values unsuitable for a production environment.

You can supply environment variables for a microservice in environment-specific *.env* files. For example, you can have an *.env.dev* file for storing environment variable values for a development environment and an *.env.ci* file for storing environment variable values used in the microservice's *continuous integration* (CI) pipeline. The syntax of *.env* files is straightforward. There is one environment variable defined per line:

.env.dev

```
NODE_ENV=development
HTTP_SERVER_PORT=3001
LOG_LEVEL=INFO
MONGODB_HOST=localhost
MONGODB_PORT=27017
MONGODB_USER=
MONGODB_PASSWORD=
```

.env.ci

```
NODE_ENV=integration
HTTP_SERVER_PORT=3001
LOG_LEVEL=INFO
MONGODB_HOST=localhost
MONGODB_PORT=27017
MONGODB_USER=
MONGODB_PASSWORD=
```

When a software component is deployed to a Kubernetes cluster using Helm, environment variable values should be defined in the Helm chart's *values.yaml* file:

values.yaml

```
nodeEnv: production
httpServer:
  port: 8080
database:
  mongoDb:
    host: my-service-mongodb
    port: 27017
```

The values in the above *values.yaml* file can be used to define environment variables in a Kubernetes Deployment using the following Helm chart template:

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-service
spec:
  template:
    spec:
      containers:
        - name: my-service
          env:
            - name: NODE_ENV
              value: {{ .Values.nodeEnv }}
            - name: HTTP_SERVER_PORT
              value: "{{ .Values.httpServer.port }}"
            - name: MONGODB_HOST
              value: {{ .Values.database.mongoDb.host }}
            - name: MONGODB_PORT
              value: {{ .Values.database.mongoDb.port }}
```

When Kubernetes starts a microservice pod, the following environment variables will be made available for the running container:

```
NODE_ENV=production
HTTP_SERVER_PORT=8080
MONGODB_HOST=my-service-mongodb
MONGODB_PORT=27017
```

Kubernetes ConfigMaps

A Kubernetes ConfigMap can store a configuration file or files in various formats, like JSON or YAML. These files can be mounted to the filesystem of a microservice's running container. The container can then read the configuration files from the mounted directory in its filesystem.

For example, you can have a ConfigMap for defining the logging level of a *my-service* microservice:

configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-service
data:
  LOG_LEVEL: INFO
```

The below Kubernetes Deployment descriptor defines that the content of the *my-service* ConfigMap's key `LOG_LEVEL` will be stored in a volume named `config-volume`, and the value of the `LOG_LEVEL` key will be stored in a file named *LOG_LEVEL*. After mounting the `config-`

`volume` to the */etc/config* directory in a *my-service* container, it is possible to read the contents of the */etc/config/LOG_LEVEL* file, which contains the text: INFO.

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-service
spec:
  template:
    spec:
      containers:
        - name: my-service
          volumeMounts:
            - name: config-volume
              mountPath: "/etc/config"
              readOnly: true
      volumes:
        - name: config-volume
          configMap:
            name: my-service
            items:
              - key: "LOG_LEVEL"
                path: "LOG_LEVEL"
```

In Kubernetes, the editing of a ConfigMap is reflected in the respective mounted file. This means that you can listen to changes in the */etc/config/LOG_LEVEL* file. Below is shown how to do it in JavaScript:

```
fs.watchFile('/etc/config/LOG_LEVEL', () => {
  try {
    const newLogLevel = fs.readFileSync(
      '/etc/config/LOG_LEVEL', 'utf-8'
    ).trim();

    // Check here that 'newLogLevel' contains a valid log level

    updateLogLevel(newLogLevel);
  } catch (error) {
    // Handle error
  }
});
```

Kubernetes Secrets

Kubernetes Secrets are similar to ConfigMaps except that they are used to store sensitive information, like passwords and encryption keys.

Below is an example of *values.yaml* file and a Helm chart template for creating a Kubernetes Secret. The Secret will contain two key-value pairs: the database username and password. The Secret's data needs to be Base64-encoded. In the below example, the Base64 encoding is done using the Helm template function `b64enc`.

values.yaml

```
database:
  mongoDb:
    host: my-service-mongodb
    port: 27017
    user: my-service-user
    password: Ak9(lKt41uF==%lLO&21mA#gL0!"Dps2
```

secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-service
type: Opaque
data:
  mongoDbUser: {{ .Values.database.mongoDb.user | b64enc }}
  mongoDbPassword: {{ .Values.database.mongoDb.password | b64enc }}
```

After being created, secrets can be mapped to environment variables in a Deployment descriptor for a microservice. In the below example, we map the value of the secret key `mongoDbUser` from the *my-service* secret to an environment variable named `MONGODB_USER` and the value of the secret key `mongoDbPassword` to an environment variable named `MONGODB_PASSWORD`.

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-service
spec:
  template:
    spec:
      containers:
        - name: my-service
          env:
            - name: MONGODB_USER
              valueFrom:
                secretKeyRef:
                  name: my-service
                  key: mongoDbUser

            - name: MONGODB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: my-service
                  key: mongoDbPassword
```

When a *my-service* pod is started, the following environment variables are made available for the running container:

```
MONGODB_USER=my-service-user
MONGODB_PASSWORD=Ak9(lKt41uF==%lLO&21mA#gL0!”Dps2
```

Service Substitution Principle

> *Make substituting a service's service dependency for another service easy by making the dependencies transparent. A transparent service is exposed to other services by defining a host and port. Use externalized service configuration principle (e.g., environment variables) in your microservice to define the host and port (and possibly other needed parameters like a database username/password) for a dependent service.*

Let's have an example where a microservice depends on a MongoDB service. The MongoDB service should expose itself by defining a host and port combination. For the microservice, you can specify the following environment variables for connecting to a *localhost* MongoDB service:

.env.dev

```
MONGODB_HOST=localhost
MONGODB_PORT=27017
```

Suppose that in a Kubernetes-based production environment, you have a MongoDB service in the cluster accessible via a Kubernetes Service named *my-service-mongodb*. In that case, you should have the environment variables for the MongoDB service defined as follows:

```
MONGODB_HOST=my-service-mongodb.default.svc.cluster.local
MONGODB_PORT=8080
```

Alternatively, a MongoDB service can run in the MongoDB Atlas cloud. Then the MongoDB service could be connected to using the following kind of environment variable values:

```
MONGODB_HOST=my-service.tjdze.mongodb.net
MONGODB_PORT=27017
```

As shown with the above examples, you can easily substitute a different MongoDB service depending on your microservice's environment. If you want to use a different MongoDB service, you don't need to modify the microservice's source code but only change the configuration.

Inter-Service Communication Methods

Services communicate with each other using the following communication methods: synchronous, asynchronous, and shared data.

Synchronous Communication Method

A synchronous communication method should be used when a service communicates with another

service and wants an immediate response. Synchronous communication can be implemented using protocols like HTTP or gRPC (which uses HTTP under the hood).

In case of a failure when processing a request, the request processing microservice sends an error response to the requestor microservice. The requestor microservice can cascade the error up in the synchronous request stack until the initial request maker is reached. Often, that initial request maker is a client, like a web or mobile client. The initial request maker can then decide what to do. Usually, it will attempt to send the request again after a while (we are assuming here that the error is a transient server error, not a client error, like a bad request, for example)

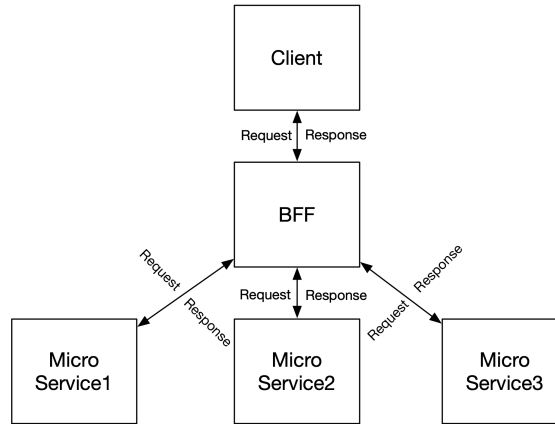

Figure 2.7 Synchronous Communication Method

Asynchronous Communication Method

When a service wants to deliver a request to another service, but does not expect a response or at least not an immediate response, then an asynchronous communication method should be used. Some communication between services is asynchronous by nature. For example, a service might want to instruct an email notification service to email an end-user or to send an audit log entry to

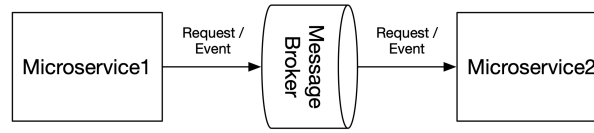

Figure 2.8 Request-Only Asynchronous Communication Method

an audit logging service. Both examples can be implemented using an asynchronous

communication method because no response for the operations is expected.

Asynchronous communication can be implemented using a message broker. Services can produce messages to the message broker and consume messages from the message broker. There are several message broker implementations available like Apache Kafka, RabbitMQ, Apache ActiveMQ and Redis. When a microservice produces a request to a message broker's topic, the producing

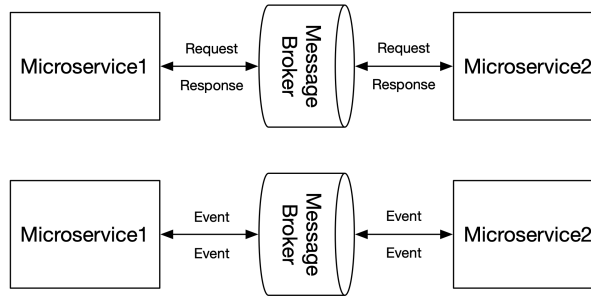

Figure 2.9 Request-Response Asynchronous Communication Method / Event Driven Architecture

microservice must wait for an acknowledgment from the message broker indicating that the request was successfully stored to multiple, or preferably all, replicas of the topic. Otherwise, there is no 100% guarantee that the request was successfully delivered in some message broker failure scenarios.

When an asynchronous request is of type fire-and-forget (i.e., no response is expected), the request processing microservice must ensure that the request will eventually get processed. If the request processing fails, the request processing microservice must reattempt the processing after a while. If a termination signal is received, the request processing microservice instance must produce the request back to the message broker and allow some other instance of the microservice to fulfill the request. The rare possibility exists that the production of the request back to the message broker fails. You could then try to save the request to a persistent volume, for instance, but also that can fail. The likelihood of such a situation is very low.

Designing APIs for inter-service communication is described in more detail in the *API design principles* chapter.

Shared Data Communication Method

Sometimes communication between services can happen via shared data (e.g., using a shared database). This method is useful with data-oriented services when storing the same data twice is not meaningful. Typically, one or more microservices produce the shared data, and other

microservice(s) consume that data. The interface between these microservices is defined by the schema of the shared data, e.g., by the schemas of database tables. To secure the shared data, only the producing microservice(s) should have write access to the shared data, and the consuming microservice(s) should only have read access to the shared data.

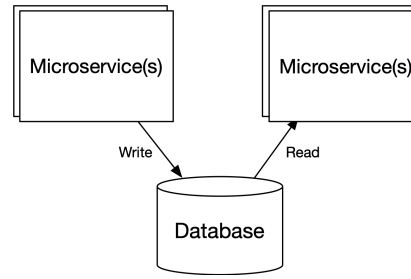

Figure 2.10 Shared Data Communication Method

Domain-Driven Architectural Design Principle

> *Design architecture by conducting domain-driven design (DDD) starting from the top of the software hierarchy (software system) and ending at the service level.*

I often compare software system architectural design to the architectural design of a house. The house represents a software system. The facade of the house represents the external interfaces of the software system. The rooms in the house are the microservices of the software system. Like a microservice, a single room usually has a dedicated purpose. The architectural design of a software system encompasses the definition of external interfaces, microservices, and their interfaces to other microservices.

The result of the architectural design phase is a ground plan for the software system. After the architectural design, you have the facade designed, and all the rooms are specified: the purpose of each room and how rooms interface with other rooms.

Designing an individual microservice is no more architectural design it is like the interior design of a single room. The design of microservices is handled using object-oriented design principles, presented in the next chapter.

Domain-driven design (DDD) is a software design approach where software is modeled to match a problem/business domain according to input from the domain experts. Usually, these experts come from the business and specifically from product management. The idea of DDD is to transfer the domain knowledge from the domain experts to individual software developers so that everyone participating in software development can share a common language that describes the domain.

The idea of the common language

is that people can understand each other, and no multiple terms are used to describe a single thing. This common language is also called the *ubiquitous language*.

The domain knowledge is transferred from product managers and architects to lead developers and product owners (POs) in development teams. The team's lead developer and PO share the domain knowledge with the rest of the team. This usually happens when the team processes epics and features and splits them into user stories in planning sessions. A software development team can also have a dedicated domain expert or experts.

DDD starts from the top business/problem domain. The top domain is split into multiple subdomains on the same abstraction level: one level lower than the top domain. A domain should be divided into subdomains so that there is minimal overlap between subdomains. Subdomains will be interfacing with other subdomains using well-defined interfaces. Subdomains are also called *bounded contexts*, and technically they represent an application or a microservice. For example, a banking software system can have a subdomain or bounded context for loan applications and another for making payments.

Design Example 1: Mobile Telecom Network Analytics Software System

Suppose an architecture team is assigned to design a mobile telecom network analytics software system. The team starts by defining the problem domain of the software system in more detail. When thinking about the system in more detail, they end up figuring out at least the following subdomains:

1. Ingesting raw data from various sources of the mobile telecom network
2. Transforming the ingested raw data into meaningful insights
3. Proper ways of presenting the insights to software system users

Let's pick up some keywords from the above definitions and formulate short names for the subdomains:

1. Ingesting raw data
2. Transforming raw data into insights
3. Presenting insights

Let's consider each of these three subdomains separately.

We know that a mobile telecom network is divided into core and radio networks. From that, we can conclude that *Ingesting raw data* domain can be divided into further subdomains: *Ingesting radio*

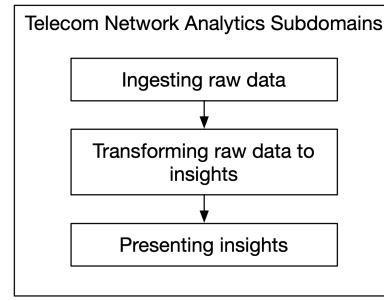

Figure 2.11 Subdomains

network raw data and *Ingesting core network raw data*. We can turn these two subdomains into applications for our software system: *Radio network data ingester* and *Core network data ingester*.

The *Transforming raw data to insights* domain should at least consist of an application aggregating the received raw data to counters and key performance indicators (KPIs). We can call that application *Data aggregator*.

The *Presenting insights* domain should contain a web application that can present insights in various ways, like using dashboards containing charts presenting aggregated counters and calculated KPIs. We can call this application *Insights visualizer*.

Now we have the following applications for the software system defined:

- Radio network data ingester
- Core network data ingester
- Data aggregator
- Insights visualizer

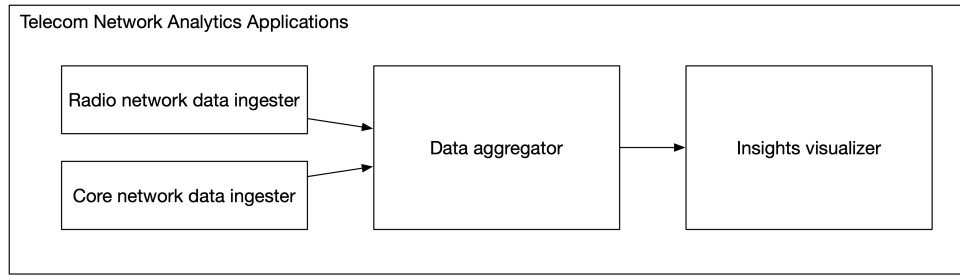

Figure 2.12 Applications

Next, we continue architectural design by splitting each application into one or more software

components. (services, clients, and libraries). When defining the software components, we must remember to follow the *single responsibility principle*, *avoid duplication principle* and *externalized service configuration principle*.

When considering the *Radio network data ingester* and *Core network data ingester* applications, we can notice that we can implement them both using a single microservice, *data-ingester-service*, with different configurations for radio and core network. This is because the protocol for ingesting the data is the same for radio and core networks. The two networks differ in the schema of the ingested data. Using a single configurable microservice, we can avoid code duplication by using externalized configuration.

The *Data aggregator* application can be implemented using a single *data-aggregator-service* microservice. We can use externalized configuration to define what counters and KPIs the microservice should aggregate and calculate.

The *Insights visualizer* application consists of three different software components:

- A web client
- A service for fetching aggregated and calculated data (counters and KPIs)
- A service for storing the dynamic configuration of the web client

The dynamic configuration service stores information about what insights to visualize and how in the web client. Microservices in the *Insights visualizer* application are:

- insights-visualizer-web-client
- insights-visualizer-data-service
- insights-visualizer-configuration-service

Now we are ready with the microservice-level architectural design for the software system.

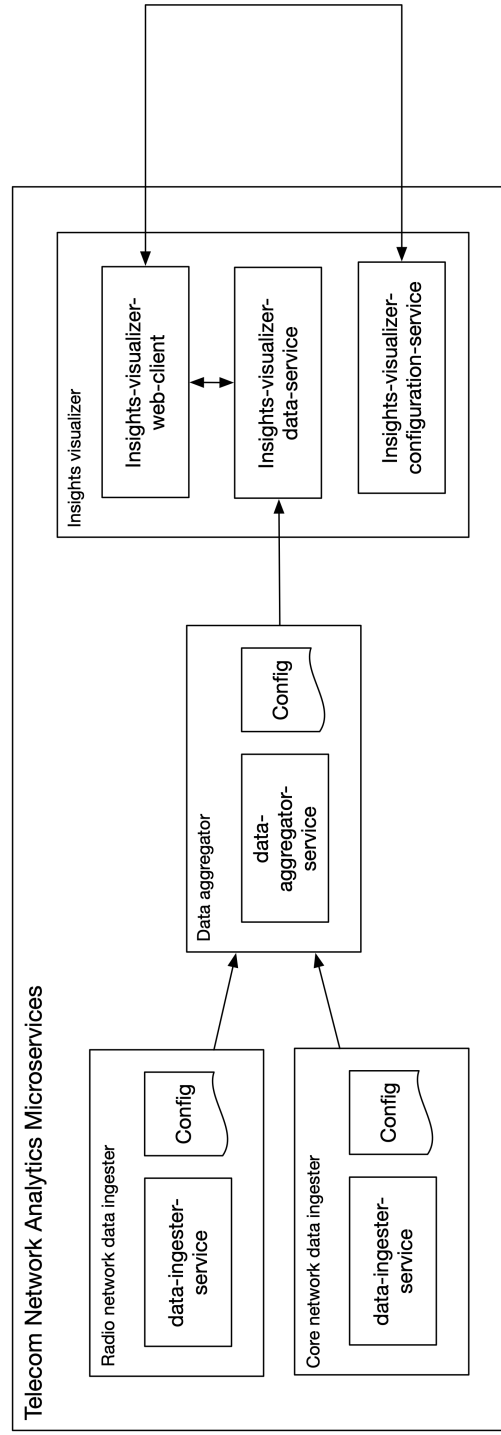

Figure 2.13 Microservices

The last part of architectural design is to define the inter-service communication methods. The *data-ingester-service* needs to send raw data to *data-aggregator-service*. The sending of data is done using asynchronous fire-and-forget requests and is implemented using a message broker. The communication between the *data-aggregator-service* and the *insights-visualizer-data-service* should use the *shared data* communication method because the *data-aggregator-service* generates aggregated data that the *insights-visualizer-data-service* uses. The communication between the *insights-visualizer-web-client* in the frontend and the *insights-visualizer-data-service* and *insights-visualizer-configuration-service* in the backend is synchronous communication that can be implemented using an HTTP-based JSON-RPC, REST, or GraphQL API.

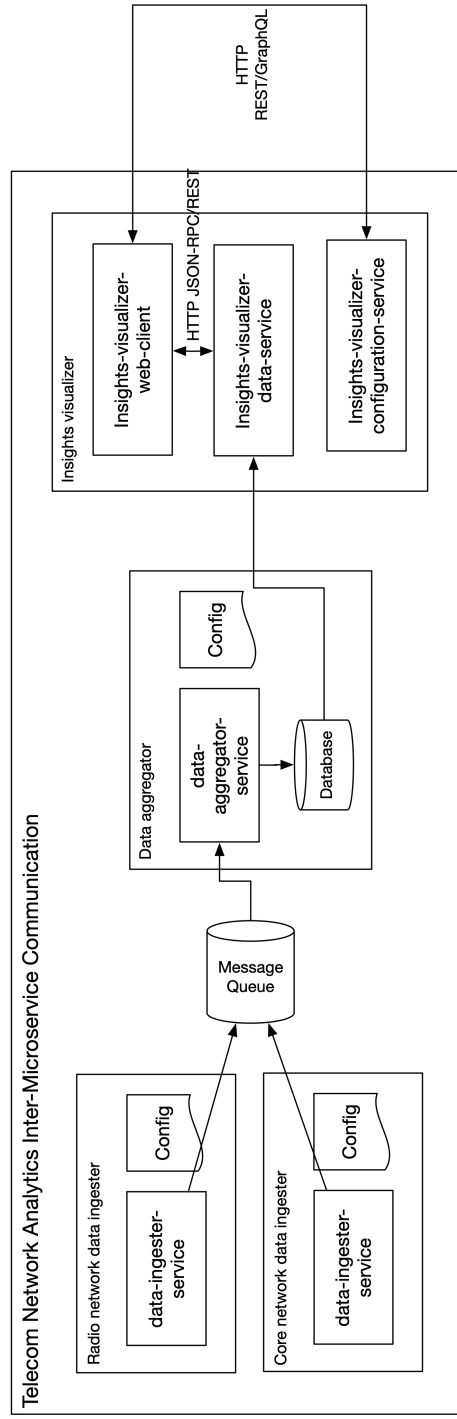

Figure 2.14 Inter-Microservice Communication

Next, design continues in development teams. Teams will specify the APIs between the microservices and conduct object oriented design for the microservices. API design is covered in a later chapter, and object-oriented design is covered in the next chapter.

Design Example 2: Banking Software System

Let's design a partial banking software system. The banking software system should be able to handle customers' loan applications and payments. The banking system problem domain can be divided into two subdomains or bounded contexts:

- Loan applications
- Making payments

In the loan applications domain, a customer can submit a loan application. The eligibility for the loan will be assessed, and the bank can either accept the loan application and pay the loan or reject the loan application. In the making payments domain, a customer can make payments. Making a payment will withdraw money from the customer's account. It is also a transaction that should be recorded.

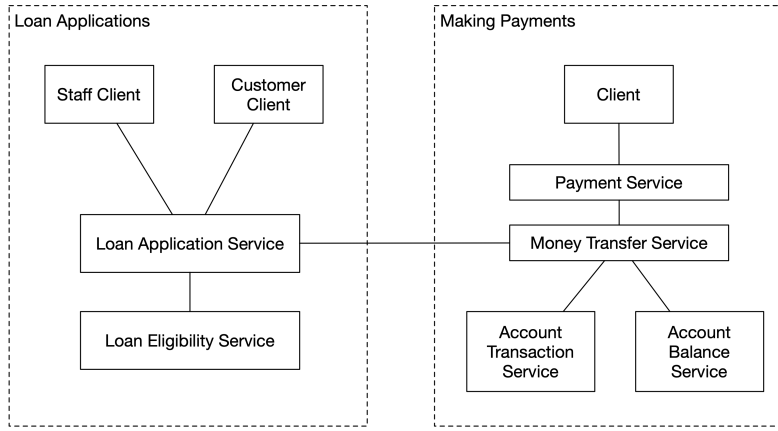

Figure 2.15 Banking Software System Bounded Contexts

Let's add a feature that a payment can be made to a recipient in another bank:

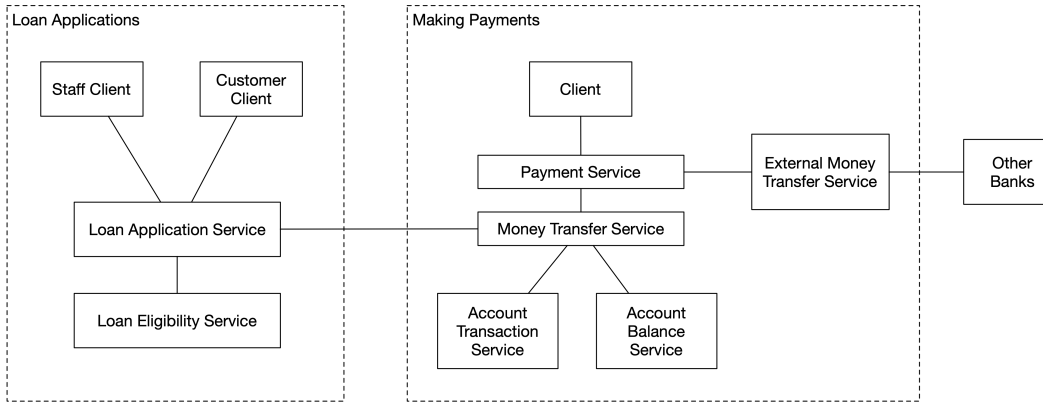

Figure 2.16 Banking Software System Bounded Contexts

Let's add another feature: money can be transferred from external banks to a customer's account.

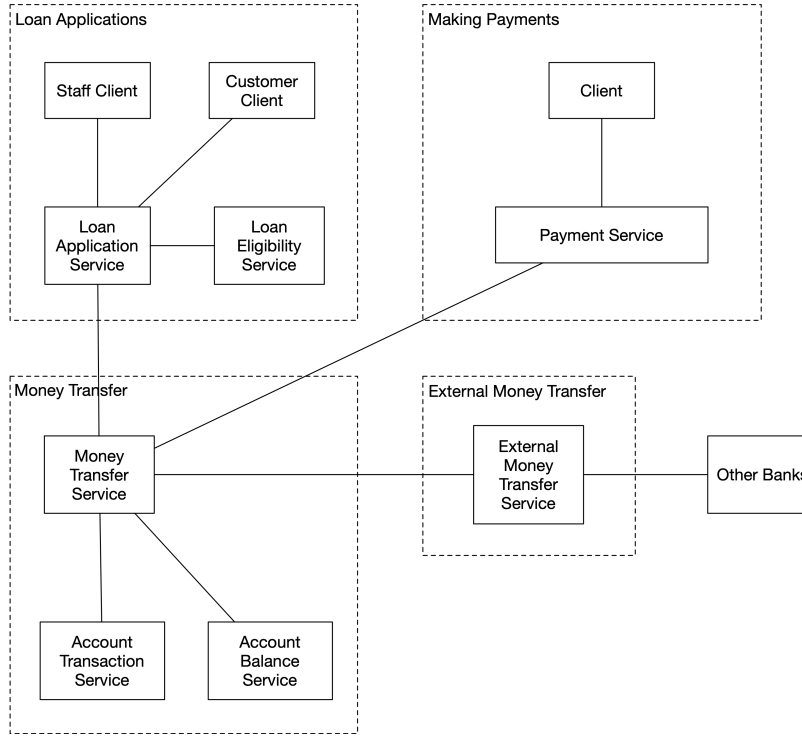

Figure 2.17 Banking Software System Bounded Contexts

As can be noticed from the above pictures, the architecture of the banking software system evolved when new features were introduced. For example, two new subdomains (or bounded contexts) were created: money transfer and external money transfer. There was not so much change in the microservices themselves, but how they are grouped logically to bounded contexts was altered.

Autopilot Microservices Principle

> *Microservices should be architected to run on autopilot in their deployment environment.*

An autopilot microservice means a microservice that runs in a deployment environment without human interaction, except in abnormal situations when the microservice should generate an alert to indicate that human intervention is required.

Autopilot microservices principle requires that the following sub-principles are followed:

- Stateless microservices principle
- Resilient microservices principle
- Horizontally autoscaling microservices principle
- Highly-available microservices principle
- Observable microservices principle

These principles are discussed in more detail next.

Stateless Microservices Principle

> *Microservices should be stateless to enable resiliency, horizontal scalability, and high availability.*

A microservice can be made stateless by storing its state outside itself. The state can be stored in a data store that microservice instances share. Typically, the data store is a database or an in-memory cache (like Redis, for example).

Resilient Microservices Principle

> *Microservices should be resilient, i.e., quickly recover from failures automatically.*

In a Kubernetes cluster, the resiliency of a microservice is handled by the Kubernetes control plane. If a computing node where a microservice instance is located needs to be decommissioned, Kubernetes will create a new instance of the microservice on another computing node and then

evict the microservice from the node to be decommissioned.

What needs to be done in a microservice is to make it listen to Linux termination signals, especially the *SIGTERM* signal, which is sent to a microservice instance to indicate that it should terminate. Upon receiving a SIGTERM signal, the microservice instance should initiate a graceful shutdown. If the microservice instance does not shut down gracefully, Kubernetes will eventually issue a *SIGKILL* signal to terminate the microservice instance forcefully. The SIGKILL signal is sent after a termination grace period has elapsed. This period is, by default, 30 seconds, but it is configurable.

There are other reasons a microservice instance might be evicted from a computing node. One such reason is that Kubernetes must assign (for some reason which can be related to CPU/memory requests, for instance) another microservice to be run on that particular computing node, and your microservice won't fit there anymore and must be moved to another computing node.

If a microservice pod crashes, Kubernetes will notice that and start a new pod so that there are always the wanted number of microservice replicas (pods/instances) running. The replica count can be defined in the Kubernetes Deployment for the microservice.

But what if a microservice pod enters a deadlock and cannot serve requests? This situation can be remediated with the help of a *liveness probe*. You should always specify a liveness probe for each microservice Deployment. Below is an example of a microservice Deployment where an HTTP GET type liveness probe is defined:

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "microservice.fullname" . }}
spec:
  replicas: 1
  selector:
    matchLabels:
      {{- include "microservice.selectorLabels" . | nindent 6 }}
  template:
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.imageRegistry }}/{{ .Values.imageRepository }}:
{{ .Values.imageTag }}"
          livenessProbe:
            httpGet:
              path: /isMicroserviceAlive
              port: 8080
            initialDelaySeconds: 30
            failureThreshold: 3
            periodSeconds: 3
```

Kubernetes will poll the */isMicroserviceAlive* HTTP endpoints of the microservice instances every three seconds (after the initial delay of 30 seconds reserved for the microservice instance startup). The HTTP endpoint should return the HTTP status code 200 *OK*. Suppose requests to that endpoint fail (e.g., due to a deadlock) three times in a row (defined by the `failureThreshold`

property) for a particular microservice instance. In that case, the microservice instance is considered dead, and Kubernetes will terminate the pod and launch a new pod automatically.

When upgrading a microservice to a newer version, the Kubernetes Deployment should be modified. A new container image tag should be specified in the `image` property of the Deployment. This change will trigger an update procedure for the Deployment. By default, Kubernetes performs a *rolling update*, which means your microservice can serve requests during the update procedure without downtime.

Suppose you had defined one replica in the microservice Deployment (as above `replicas: 1`), and performed a Deployment upgrade (change the image to a newer version). In that case, Kubernetes would create a new pod using the new image tag, and only after the new pod is ready to serve requests will Kubernetes delete the pod running the old version. So there is no downtime, and the microservice can serve requests during the upgrade procedure.

If your microservice deployment had more replicas, e.g., 10, by default, Kubernetes would terminate max 25% of the running pods and start max 25% of the replica count new pods. The *rolling update* means that updating pods happens in chunks, 25% of the pods at a time. The percentage value is configurable.

Horizontally Autoscaling Microservices Principle

Microservices should automatically scale horizontally to be able to serve more requests.

Horizontal scaling means adding new instances or removing instances of a microservice. Horizontal scaling of a microservice requires statelessness. Stateful services are usually implemented using sticky sessions so that requests from a particular client go to the same service instance. The horizontal scaling of stateful services is complicated because a client's state is stored on a single service instance. In the cloud-native world, we want to ensure even load distribution between microservice instances and target a request to any available microservice instance for processing.

Initially, a microservice can have one instance only. When the microservice gets more load, one instance cannot necessarily handle all the work. In that case, the microservice must be scaled horizontally (scaled out) by adding one or more new instances. When several microservice instances are running, the state cannot be stored inside the instances anymore because different client requests can be directed to different microservice instances. A stateless microservice must store its state outside the microservice in an in-memory cache or a database shared by all the microservice instances.

Microservices can be scaled manually, but that is rarely desired. Manual scaling requires someone to constantly monitor the software system and make the needed scaling actions manually. Microservices should scale horizontally automatically. There are two requirements for a microservice to be horizontally auto-scalable:

- Microservice must be stateless
- There must be one or more metrics that define the scaling behavior

Typical metrics for horizontal autoscaling are CPU utilization and memory consumption. In many cases, using the CPU utilization metric alone can be enough. It is also possible to use a custom or external metric. For example, the Kafka consumer lag metric can indicate if the consumer lag is increasing and if a new microservice instance should be spawned to reduce the consumer lag.

In Kubernetes, you can specify horizontal autoscaling using the *HorizontalPodAutoscaler* (HPA):

hpa.yaml

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: my-service
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-service
  minReplicas: 1
  maxReplicas: 99
  metrics:
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: 75
    - type: Resource
      resource:
        name: memory
        targetAverageUtilization: 75
```

In the above example, the *my-service* microservice is horizontally auto-scaled so that there is always at least one instance of the microservice running. There can be a maximum of 99 instances of the microservice running. The microservice is scaled out if CPU or memory utilization is over 75%, and it is scaled in (the number of microservice instances is reduced) when both CPU and memory utilization falls below 75%.

Highly-Available Microservices Principle

> *Business-critical microservices must be highly-available.*

If only one microservice instance runs in an environment, it does not make the microservice highly available. If something happens to that one instance, the microservice becomes temporarily unavailable until a new instance has been started and is ready to serve requests. For this reason, you should run at least two or more instances for all business-critical microservices. You should also ensure that these two instances don't run on the same computing node. The instances should run in different availability zones of the cloud provider. Then a catastrophe in availability zone 1

won't necessarily affect microservices running in availability zone 2.

You can ensure that no two microservice instances run on the same computing node by defining an anti-affinity rule in the microservice Deployment:

deployment.yaml

```
.
.
.
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchLabels:
            app.kubernetes.io/name: {{ include "microservice.name" . }}
        topologyKey: "kubernetes.io/hostname"
.
.
.
```

For a business-critical microservice, we need to modify the horizontal autoscaling example from the previous section: The `minReplicas` property should be increased to 2:

hpa.yaml

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: my-service
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-service
  minReplicas: 2
  maxReplicas: 99
  .
  .
  .
```

Observable Microservices Principle

> *It should be possible to detect any abnormal behavior in deployed microservices as soon as possible. Abnormal behavior should trigger an alert. The deployment environment should offer aids for troubleshooting abnormal behavior.*

A modern cloud-native software system consists of multiple microservices running simultaneously. No one can manually check the logs of tens or hundreds of microservice instances. The key to monitoring microservices is automation. Everything starts with collecting relevant metrics from microservices and their execution environment. These metrics are used to define automatic alerting of abnormal conditions. Metrics are also used to create monitoring and troubleshooting dashboards, which can be used to analyze the state of the software system and its microservices

after an alert is triggered.

In addition to metrics, to enable drill-down to a problem's root cause, distributed tracing should be implemented to log the communication between different microservices to troubleshoot inter-service communication problems. Each microservice must also log at least all errors and warnings. These logs should be fed to a centralized log collection system where querying the logs is made easy.

Software Versioning Principles

In this section, the following principles related to software versioning will be presented:

- Use semantic versioning
- Avoid using 0.x versions
- Don't increase major versions
- Implement security patches and bug corrections to all major versions
- Avoid using non-LTS (Long Term Support) versions in production

Use Semantic Versioning Principle

> *Use semantic versioning for software components.*

Semantic versioning means that given a version number in the format: `<MAJOR>.<MINOR>.<PATCH>`, increment the:

- *MAJOR* value when you make incompatible API changes
- *MINOR* value when you add functionality in a backward-compatible manner
- *PATCH* value when you make backward-compatible bug fixes or security patches

Avoid Using 0.x Versions Principle

> *If you are using 3rd party components, avoid using 0.x versioned components.*

In semantic versioning, major version zero (0.x.y) is for initial development. Anything can change at any time. The public API should not be considered stable. Typically, software components with zero major versions are still in a proof of concept phase, and anything can change. If you want or need to take a newer version into use, you must be prepared for changes, and sometimes these changes can be considerable, resulting in a lot of refactoring.

Don't Increase Major Version Principle

When making backward-incompatible public API changes, you need to increase the major version. I advise not to make backward-incompatible changes, thus no major version increases.

If you need to make a backward-incompatible public API change, you should create a totally new software component with a different name. For example, suppose you have a *common-ui-library* and need to make backward-incompatible changes. In that case, it is recommended to add the new major version number to the library name and publish a new library, *common-ui-library-2*. This protects developers from accidentally using a more recent non-compatible version when changing the used library version number. Library users don't necessarily know if a library uses semantic versioning properly or not. This information is not usually told in the library documentation, but it is a good practice to communicate that in the library documentation.

If a software component is using the *common-ui-library*, it can safely always take the latest version of the library into use which contains all the needed bug fixes and security updates. If you were using Node.js and NPM, this would be safe:

```
npm install --save common-ui-library@latest
```

And when you are ready to migrate to the new major version of the library, you can uninstall the old version and install the new major version in the following way:

```
npm uninstall common-ui-library
npm install --save common-ui-library-2
```

Consider when to create a new major version of a library. When you created the first library version, you probably did not get everything right in the public API. That is normal. It is almost impossible to create a perfect API the first time. Before releasing the second major version of the library, I suggest reviewing the new API with a team, collecting user feedback, and waiting long enough to get the API "close to perfect" the second time. No one wants to use a library with frequent backward-incompatible major version changes.

Implement Security Patches and Bug Corrections to All Major Versions Principle

If you have authored a library for others to use, do not force the users to take a new major version of the library into use just because it contains some bug corrections or security patches that are not available for the older major version(s). You should have a comprehensive set of automated tests to ensure that a bug fix or security patch doesn't break anything. Thus, making a security patch or bug fix in multiple branches or source code repositories should be easy.

Requiring library users to upgrade to a new major version to get some security patch or a bug

correction can create a maintenance hell where the library users must refactor all software components using the library just to get a security patch or bug correction.

Avoid Using Non-LTS Versions in Production Principle

Some software is available as Long Term Support (LTS) and non-LTS versions. Always use only an LTS version in production. You are guaranteed long-term support through bug corrections and security patches. You can use a non-LTS version for proof of concept projects where you want to use some new features unavailable in an LTS version. But you must remember that if the PoC succeeds, you can't just throw it into production. You need to productize it first, i.e., replace the non-LTS software with LTS software.

Git Version Control Principle

> *Develop software in feature branches that are merged into the main branch. Use feature toggles when needed.*

When you need to develop a new feature, it can be done using either of the following ways:

1. Using a feature branch
2. Using multiple feature branches and a feature toggle

Feature Branch

The feature branch approach is enough for simple features encompassing a single program increment, team, and microservice. A new feature is developed in a feature branch created from the main branch, and when the feature is ready, the feature branch is merged back into the main branch, and the feature branch can be deleted if wanted. The feature branch should be merged using a merge or pull request that triggers a CI pipeline run that must succeed before the merge/pull request can be completed. The merge or pull request should also take care of the code review. There should also be a manual way to trigger a CI/CD pipeline run for a feature branch so that developers can test an unfinished feature in a test environment during the development phase.

Below a sample workflow of creating and using a feature branch is depicted:

```
git clone <repository-url>
git checkout main
git pull origin main

# Create and checkout a feature branch named "feature-id"
git checkout -b <feature-id>
```

```
# First commit
git commit -a -m "Commit message"

# More commits...

# Push feature branch
git push origin <feature-id>

# Other developers can now also use the feature branch
# because it is pushed to origin
```

When the feature is ready, you can create a pull or merge request from the feature branch to the main branch.

Feature Toggle

A feature toggle is similar to a feature license. In the case of a feature license, the feature is available only when a user has the respective license activated in their environment. A toggleable feature is available only when the feature toggle is switched on. Feature toggles should be used for complex features spanning multiple program increments, microservices, or teams. Feature toggles are part of the configuration of an environment. For example, feature toggles can be stored in a Kubernetes ConfigMap that any microservice can access. When using a feature toggle, the toggle is initially switched off. Development of the feature happens in multiple feature branches in different teams. Teams merge their part of the feature to the main branch. When all feature branches are merged into the main branch, the feature toggle can be switched on to activate the feature.

Architectural Patterns

Event Sourcing Pattern

> *Use event sourcing to capture state changes as a sequence of events.*

Event sourcing ensures that all changes to the state of a service are stored as an ordered sequence of events. Event sourcing makes it possible to query state changes. Also, the state change events act as an audit log. It is possible to reconstruct past states and rewind the current state to some earlier state. Unlike CRUD actions on resources, event sourcing utilizes only CR (create and read) actions. It is only possible to create new events and read events. It is not possible to update an existing event or delete an event.

Let's have an example of using event sourcing to store orders in an e-commerce software system. The *order-service* should be able to store the following events:

- AbstractOrderEvent
    - Abstract base event for other concrete events containing timestamp and order id properties
- OrderCreatedEvent
    - Contains basic information about the order
- OrderPaymentEvent
    - Contains information about the order payment
- OrderModificationEvent
    - Contains information about modifications made by the customer to the order before packaging
- OrderPackagedEvent
    - Contains information about who collected and packaged the order
- OrderCanceledEvent
    - Describes that the customer has canceled the order and the order should not be shipped
- OrderShippedEvent
    - Contains information about the logistics partner and the tracking id of the order shipment
- OrderDeliveredEvent
    - Contains information about the pick-up point of the delivered order
- OrderShipmentReceivedEvent
    - Informs that the customer has received the shipment
- OrderReturnedEvent
    - Contains information about the returned order or order item(s)
- OrderReturnShippedEvent
    - Contains information about the logistics partner and the tracking id of the return shipment
- OrderReturnReceivedEvent
    - Contains information about who handled the order return and the status of returned items
- OrderReimbursedEvent
    - Contains information about the reimbursement for the returned order item(s) to the customer

Command Query Responsibility Segregation (CQRS) Pattern

> *Use the CQRS pattern if you want to use a different model for create/update (= command) operations compared to the model you want to use to query information.*

Let's consider the previous *order-service* example that used event sourcing. In the *order-service*,

all the commands are events. We want users to be able to query orders efficiently. We should have an additional representation of an order in addition to events because it is inefficient to always generate the current state of an order by replaying all the related events. For this reason, our architecture should utilize the CQRS pattern and divide the *order-service* into two different services: *order-command-service* and *order-query-service*.

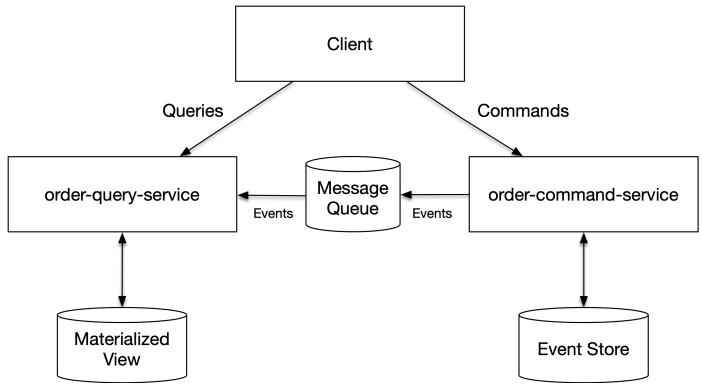

Fig 2.14 Order Services using CQRS

The *order-command-service* is the same as the original *order-service* that uses event sourcing, and the *order-query-service* is a new service. The *order-query-service* has a database where it holds a materialized view of orders. The two services are connected with a message broker. The *order-command-service* sends events to a topic in the message broker. The *order-query-service* reads events from the topic and applies changes to the materialized view. The materialized view is optimized to contain basic information about each order, including its current state, to be consumed by the e-commerce company staff and customers. Because customers query orders, the materialized view should be indexed by the customer id column to enable fast retrieval. Suppose that, in some special case, a customer needs more details about an order that is available in the materialized view. In that case, the *order-command-service* can be used to query the events of the order for additional information.

Distributed Transaction Patterns

A distributed transaction is a transaction that spans multiple microservices. A distributed transaction consists of one or more remote requests. Distributed transactions can be implemented using the *saga pattern*. In the saga pattern, each request in a distributed transaction should have a respective compensating action defined. If one request in the distributed transaction fails, compensating requests should be executed for the already successfully executed requests. The idea of executing the compensating requests is to bring the system back to the state where it was before the distributed transaction was started. So, the rollback of a distributed transaction is done via

executing the compensating actions.

A failed request in a distributed transaction must be conditionally compensated if we cannot be sure whether the server successfully executed the request. This can happen when a request timeouts and we don't receive a response to indicate the request status.

Also, executing a compensating request can fail. For this reason, a microservice must persist compensating requests so they can be retried later until they all succeed. Persistence is needed because the microservice instance can be terminated before it has completed all the compensating requests successfully. Another microservice instance can continue the work left by the terminated microservice instance.

Some requests in a distributed transaction can be such that they cannot be compensated. One typical example is sending an email. You can't get it unsent once it has been sent. There are at least two approaches to dealing with requests that cannot be compensated. The first one is to delay the execution of the request so that it can be made compensable. For example, an email-sending microservice can, instead of immediately sending an email, store the email in a queue for later sending. Now email-sending microservice can accept a compensating request to remove the email from the sending queue.

Another approach is to execute non-compensable requests in the latest possible phase of the distributed transaction. You can, for example, issue the email-sending request as the last request of the distributed transaction. Then the likelihood of needing to compensate for the email sending is lower than if the email was sent as the first request in the distributed transaction. You can also combine these two approaches. Sometimes a request can be compensable even if you first think it is not. If we think about sending an email, it could be compensated by sending another email, where you state that the earlier sent email should be disregarded for a specific reason.

Saga Orchestration Pattern

> *Orchestrator or controller microservice orchestrates the execution of a distributed transaction.*

Let's have an example of a distributed transaction using the saga orchestration pattern with an online banking system where users can transfer money from their accounts. We have a higher-level microservice called *money-transfer-service*, which is used to make money transfers. The banking system has also two lower-level microservices called *account-balance-service* and *account-transaction-service*. The *account-balance-service* holds accounts' balance information while the *account-transaction-service* keeps track of all transactions on the accounts. The *money-transfer-service* acts as a saga orchestrator and utilizes both of the lower-level microservices to make a money transfer to happen.

Let's consider a distributed transaction executed by the *money-transfer-service* when a user makes a withdrawal of $25,10:

1) The *money-transfer-service* tries to withdraw the amount from the user's account by sending the following request to the *account-balance-service*:

```
POST /account-balance-service/accounts/123456789012/withdraw
{
  "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
  "amountInCents": 2510
}
```

The `sagaUuid` is a universally unique identifier (UUID) generated by the saga orchestrator before the saga begins. If there are not enough funds to withdraw the given amount, the request fails with the HTTP status code 400 *Bad Request*. If the request is successfully executed, the *account-balance-service* should store the saga UUID to a database table temporarily. This table should be cleaned up regularly by deleting old enough saga UUIDs.

2) The *money-transfer-service* will create a new account transaction for the user's account by sending the following request to the *account-transaction-service*:

```
POST /account-transaction-service/accounts/123456789012/transactions
{
  "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
  // Additional transaction information here...
}
```

The above-described distributed transaction has two requests, each of which can fail. Let's consider the scenario where the first request to the *account-balance-service* fails. If the first request fails due to a request timeout, we don't know if the request was successfully processed by the recipient microservice. We don't know because we did not get the response and status code. For that reason, we need to perform a compensating action by issuing the following compensating request:

```
POST /account-balance-service/accounts/123456789012/undo-withdraw
{
  "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
  "amountInCents": 2510
}
```

The *account-balance-service* will perform the `undo-withdraw` action only if a withdrawal with the given saga UUID was earlier made and that withdrawal has not been undone yet. Upon successful undoing, the *account-balance-service* will delete the row for the given saga UUID from the database table where the saga UUID was earlier temporarily stored. Further `undo-withdraw` actions with the same saga UUID will be no-op actions making the `undo-withdraw` action idempotent.

Next, let's consider the scenario where the first request succeeds and the second request fails. Now we have to compensate for both requests. First, we compensate for the first request as described earlier. Then we will compensate for the second request by deleting the account transaction identified with the `sagaUuid`:

```
DELETE /account-transaction-service/accounts/123456789012/transactions?
sagaUuid=e8ab60b5-3053-46e7-b8da-87b1f46edf34
```

If a compensating request fails, it must be repeated until it succeeds. Notice that the above compensating requests are both idempotent, i.e., they can be executed multiple times with the same result. Idempotency is a requirement for a compensating request because it can be possible that a compensating request fails after the compensation was already performed. That compensation request failure will cause the compensating request to be attempted again. The distributed transaction manager in the *money-transfer-service* should ensure that a distributed transaction is successfully completed or roll-backed by the instances of the *money-transfer-service*. You should implement a single distributed transaction manager library per programming language or technology stack and use that in all microservices that need to orchestrate distributed transactions. Alternatively, use a 3rd party library.

Let's have another short example with the *ecommerce-service* presented earlier in this chapter. The order-placing endpoint of the *ecommerce-service* should make the following requests in a distributed transaction:

1. Ensure payment
2. Create an order
3. Remove the ordered sales items from the shopping cart
4. Mark the ordered sales items sold
5. Enqueue an order confirmation email for sending

The respective compensating requests are the following:

1. Reimburse the payment
2. Delete the order using the saga UUID
3. Add the ordered sales items back to the shopping cart. (The shopping cart service must ensure that a sales item can be added only once to a shopping cart)
4. Mark the ordered sales items for sale
5. Dequeue the order confirmation email

Saga Choreography Pattern

> *Microservices perform a distributed transaction in a choreography where a client microservice initiates a distributed transaction, and the last microservice involved completes the distributed transaction by sending a completion message to the client microservice.*

The saga choreography pattern utilizes asynchronous communication between microservices. Involved microservices send messages to each other in a choreography to achieve saga completion.

The saga choreography pattern has a couple of drawbacks:

- The execution of a distributed transaction is not centralized like in the saga orchestration pattern, and it can be hard to figure out how a distributed transaction is actually performed.
- It creates coupling between microservices, while microservices should be as loosely coupled as possible.

The saga choreography pattern works best in cases where the number of participating microservices is low. Then the coupling between services is low, and it is easier to reason how a distributed transaction is performed.

Let's have the same money transfer example as earlier, but now using the saga choreography pattern instead of the saga orchestration pattern.

1) The *money-transfer-service* initiates the saga by sending the following event to the message broker's *account-balance-service* topic:

```
{
  "event": "Withdraw",
  "data": {
    "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
    "amountInCents": 2510,
    // Additional transaction information here...
  }
}
```

2) The *account-balance-service* will consume the `Withdraw` event from the message broker, perform a withdrawal, and if successful, send the same event to the message broker's *account-transaction-service* topic.

3) The *account-transaction-service* will consume the `Withdraw` event from the message broker, persist an account transaction, and if successful, send the following event to the message broker's *money-transfer-service* topic:

```
{
  "event": "WithdrawComplete",
  "data": {
    "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34"
  }
}
```

If either step 2) or 3) fails, the *account-balance-service* or *account-transaction-service* will send the following event to message broker's *money-transfer-service* topic:

```
{
  "event": "WithdrawFailure",
  "data": {
    "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34"
  }
}
```

If the *money-transfer-service* receives a `WithdrawFailure` event or does not receive a `WithdrawComplete` event during some timeout period, the *money-transfer-service* will initiate a distributed transaction rollback sequence by sending the following event to the message broker's *account-balance-service* topic:

```
{
  "event": "WithdrawRollback",
  "data": {
    "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
    "amountInCents": 2510,
    // Additional transaction information here...
  }
}
```

Once the rollback in the *account-balance-service* is done, the rollback event will be produced to the *account-transaction-service* topic in the message broker. After the *account-transaction-service* has successfully performed the rollback, it sends a `WithdrawRollbackComplete` event to the *money-transfer-service* topic. Once the *money-transfer-service* consumes that message, the withdrawal event is successfully rolled back. Suppose the *money-transfer-service* does not receive the `WithdrawRollbackComplete` event during some timeout period. In that case, it will restart the rollback choreography by resending the `WithdrawRollback` event to the *account-balance-service*.

Preferred Technology Stacks Principle

> *Define preferred technology stacks for different purposes.*

The microservice architecture enables using the most suitable technology stack to develop each microservice. For example, some microservices require high performance and controlled memory allocation, and other microservices don't need such things. You can choose the used technology stack based on the needs of a microservice. For a real-time data processing microservice, you might pick C++ or Rust, and for a simple REST API, you might choose Node.js and Express, Java and Spring Boot, or Python and Django.

Even if the microservice architecture allows different teams and developers to decide what programming languages and technologies to use when implementing a microservice, defining preferred technology stacks for different purposes is still a good practice. Otherwise, you might find yourself in a situation where numerous programming languages and technologies are used in a software system. Some programming languages and technologies like Clojure, Scala, or Haskell can be relatively niche. When software developers in the organization come and go, you might end up in situations where you don't have anyone who knows about some specific niche programming language or technology. In the worst case, a microservice needs to be reimplemented from scratch using some more mainstream technologies. For this reason, you should specify technology stacks

that teams should use. These technology stacks should contain as much as possible mainstream programming languages and technologies.

For example, an architecture team might decide the following:

- Web clients should be developed using TypeScript, React, and Redux
- Non-API backend services should be developed in C++ for performance reasons
- APIs should be developed with TypeScript, Node.js, and Nest.js or with Java and Spring Boot
- Integration tests should be implemented with Cucumber using the same language as is used for the implementation or, alternatively, with Python and Behave
- E2E tests should be implemented with Python and Behave
- Scripts should be implemented using Bash for small scripts and Python for larger scripts

The above technology stacks are mainstream. Recruiting talent with needed knowledge and competencies should be effortless.

After you have defined the preferred technology stacks, you should create a utility or utilities that can be used to kick-start a new project using a particular technology stack quickly. This utility or utilities should generate the initial source code repository content for a new microservice, client, or library. The initial source code repository should contain at least the following items for a new microservice:

- Source code folder
- Unit test folder
- Integration test folder
- Build tools, like Gradle Wrapper for Java, for example
- Initial build definition file(s), like build.gradle for Java, CMakeLists.txt for C++ or package.json for Node.js
    - Initial dependencies defined in the build definition file
- .env file(s) to store environment variables for different environments (dev, CI)
- .gitignore
- README.MD template
- Linting rules (e.g., .eslintrc.json)
- Code formatting rules (e.g., .prettier.rc)
- Initial code for integration tests, e.g., docker-compose.yml file for spinning up an integration testing environment
- Infrastructure code for the chosen cloud provider, e.g., code to deploy a managed SQL database in the cloud
- Code (e.g., Dockerfile) for building the microservice container image
- Deployment code (e.g., a Helm chart)

- CI/CD pipeline definition code

The utility should ask the following questions from the developer before creating the initial source code repository content for a microservice:

- What is the name of the microservice?
- To what cloud environment will microservice be deployed? (AWS, Azure, Google Cloud, etc.)
- What are the used inter-service communication methods? Based on the answer, the utility can add dependencies, e.g., a Kafka client dependency
- Should microservice have a database, and what database?
- What are the other dependent microservices?

Of course, decisions about the preferred technology stacks are not engraved in stone. They are not static. As time passes, new technologies arise, and new programming languages gain popularity. At some point, a decision could be made that a new technology stack should replace an existing preferred technology stack. Then all new projects should use the new stack, and old software components will be gradually migrated to use the new technology stack.

Many developers are keen on learning new things on a regular basis. They should be encouraged to work on hobby projects with technologies of their choice, and they should be able to utilize new programming languages and frameworks in selected new projects.

Object-Oriented Design Principles

This chapter describes principles related to object-oriented design. The following principles are discussed:

- SOLID principles
- Uniform naming principle
- Encapsulation principle
- Composition principle
- Domain-driven design principle
- Use the design patterns principle
- Don't ask, tell principle
- Law of Demeter
- Avoid primitive type obsession principle
- Dependency injection principle
- Avoid duplication principle

SOLID Principles

All five SOLID principles (https://en.wikipedia.org/wiki/SOLID) are covered in this section. The *dependency inversion principle* is generalized as a *program against interfaces principle*. The five SOLID principles are the following:

- **S**ingle responsibility principle
- **O**pen-closed principle
- **L**iskov's substitution principle
- **I**nterface segregation principle

- **D**ependency inversion principle (Generalization: program against interfaces principle)

Single Responsibility Principle

> *Classes should have one responsibility, representing a thing or providing a single functionality. Functions should do one thing only.*

Each class should have a single dedicated purpose. A class can represent a single thing, like a bank account (`Account` class) or an employee (`Employee` class), or provide a single functionality like parsing a configuration file (`ConfigFileParser` class) or calculating tax (`TaxCalculator` class).

We should not create a class representing a bank account and an employee. It is simply wrong. Of course, an employee can *have* a bank account. But that is a different thing. It is called object composition. In object composition, an `Employee` class object contains an `Account` class object. The `Employee` class still represents one thing: An employee (who can have a bank account). Object composition is covered later in this chapter in more detail.

At the function level, each function should perform a single task. The function name should describe what task the function performs, meaning each function name should contain a verb. The function name should not contain the word *and* because it can mean that the function is doing more than one thing or you haven't named the function on a correct abstraction level. You should not name a function according to the steps it performs (e.g., *doThisAndThatAndThenThirdThing*) but instead, use wording on a higher level of abstraction.

When a class represents something, it can contain multiple methods. For example, in the `Account` class, there can be methods like `deposit` and `withdraw`. It is still a single responsibility if these methods are simple enough and if there are not too many methods in the class.

Below is a real-life code example where the *and* word is used in the function name:

```
void deletePageAndAllReferences(Page page) {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
```

In the above example, the function seems to do two things: deleting a page and removing all the references to that page. But if we look at the code inside the function, we can realize that it is doing a third thing also: deleting a page key from configuration keys. So should the function be named `deletePageAndAllReferencesAndConfigKey`? It does not sound reasonable. The problem with the function name is that it is at the same level of abstraction as the function statements. The function name should be at a higher level of abstraction than the statements inside the function.

How should we then name the function? I cannot say for sure because I don't know the context of the function. We could name the function just `delete`. This would tell the function caller that a page will be deleted. The caller does not need to know all the actions related to deleting a page. The caller just wants a page to be deleted. The function implementation should fulfill that request and do the needed housekeeping actions, like removing all the references to the page being deleted and so on.

Another example of a function doing multiple things is JavaScript's `splice` method in the `Array` class. The function name should describe what it does (more about that in the *uniform naming principle* section). But nobody would know what a `splice` function does if they just heard the name for the first time. If the function name cannot tell what the function does, the function might do more than one thing. The description of the `splice` method tells the following:

> *The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.*

The above description indicates that the `splice` method does multiple things: removing, replacing, and adding. This method should be split into multiple methods, each having a single responsibility of removing, replacing, or adding.

Let's consider another example with React Hooks. React Hooks have a function named `useEffect` which can be used to enqueue functions to be run after component rendering. The `useEffect` function can be used to run some code after the initial render (after the component mount), after every render, or conditionally. This is quite much responsibility for a single function. Also, the function's quite strange name does not reveal its purpose. The word *effect* comes from the fact that this function is used to enqueue other functions with side effects to be run. The term *side effect* might be familiar to functional language programmers. It indicates that a function is not pure (has side effects).

Below is an example React functional component:

MyComponent.jsx

```
import { useEffect } from "react";

export default function MyComponent() {
  useEffect(() => {
    function startFetchData() {
      // ...
    }

    function subscribeToDataUpdates() {
      // ...
    }

    function unsubscribeFromDataUpdates() {
      // ...
    }

    startFetchData();
```

```
    subscribeToDataUpdates();
    return function cleanup() { unsubscribeFromDataUpdates() };
  }, []);

  // JSX to render
  return ...;
}
```

In the above example, the `useEffect` call makes calls to functions `startFetchData` and `subscribeToDataUpdates` to happen after the initial render because of the supplied empty array for dependencies (the second parameter to the `useEffect` function). The cleanup function returned from the function supplied to `useEffect` will be called before the effect will be run again or when the component is unmounted and in this case, only on unmount because the effect will only run once after the initial render.

Let's imagine how we could improve the `useEffect` function. We could separate the functionality related to mounting and unmounting into two different functions: `afterMount` and `beforeUnmount`. Then we could change the above example to the following piece of code:

```
export default function MyComponent() {
  function startFetchData() {
    // ...
  }

  function subscribeToDataUpdate() {
    // ...
  }

  function unsubscribeFromDataUpdate() {
    // ...
  }

  afterMount(startFetchData, subscribeToDataUpdates);
  beforeUnmount(unsubscribeFromDataUpdates)

  // JSX to render
  return ...;
}
```

The above example is cleaner and much easier for a reader to understand than the original example. There are no multiple levels of nested functions. You don't have to return a function to be executed on component unmount, and you don’t have to supply an array of dependencies.

Let's have another example of a React functional component:

```
import { useEffect, useState } from "react";

export default function ButtonClickCounter() {
  const [clickCount, setClickCount] = useState(0);

  useEffect(() => {
    function updateClickCountInDocumentTitle() {
      document.title = `Click count: ${clickCount}`;
    }
```

```
    updateClickCountInDocumentTitle();
  });
}
```

In the above example, the effect is called after every render (because no dependencies array is supplied for the `useEffect` function). Nothing in the above code clearly states what will be executed and when. We still use the same `useEffect` function, but now it behaves differently compared to the previous example. It seems like the `useEffect` function is doing multiple things. How to solve this? Let's think hypothetically again. We could introduce yet another new function that can be called when we want something to happen after every render:

```
export default function ButtonClickCounter() {
  const [clickCount, setClickCount] = useState(0);

  afterEveryRender(function updateClickCountInDocumentTitle() {
    document.title = `Click count: ${clickCount}`;
  });
}
```

The intentions of the above React functional component are pretty clear: It will update the click count in the document title after every render.

Let's optimize our example so that the click count update happens only if the click count has changed:

```
import { useEffect, useState } from "react";

export default function ButtonClickCounter() {
  const [clickCount, setClickCount] = useState(0);

  useEffect(() => {
    function updateClickCountInDocumentTitle() {
      document.title = `Click count: ${clickCount}`;
    }

    updateClickCountInDocumentTitle();
  }, [clickCount]);
}
```

Notice how `clickCount` is now added to the dependencies array of the `useEffect` function. This means the effect is not executed after every render but only when the click count is changed.

Let's imagine how we could improve the above example. We could introduce a new function that handles dependencies: `afterEveryRenderIfChanged`. Our hypothetical example would now look like this:

```
export default function ButtonClickCounter() {
  const [clickCount, setClickCount] = useState(0);

  afterEveryRenderIfChanged(
    [clickCount],
```

```
    function updateClickCountInDocumentTitle() {
      document.title = `Click count: ${clickCount}`;
  });
}
```

Making functions do a single thing also helped make the code more readable. Regarding the original examples, a reader must look at the end of the `useEffect` function call to figure out in what circumstances the effect function will be called. And it is cognitively challenging to understand and remember the difference between a missing and empty dependencies array. Good code is such that it does not make the code reader think. At best, code should read like prose: *after every render if changed "clickCount", update click count in document title*.

One idea behind the single responsibility principle is that it enables software development using the *open-closed principle* described in the next section. When you follow the single responsibility principle and need to add functionality, you add it to a new class, which means you don't need to modify an existing class. You should avoid modifying existing code but extend it by adding new classes, each with a single responsibility.

Open-Closed Principle

> *Software code should be open for extension and closed for modification. Functionality in existing classes should not be modified, but new classes should be introduced that either implement a new or existing interface or extend an existing class.*

Any time you find yourself modifying some method in an existing class, you should first consider if this principle could be followed and if the modification could be avoided. Every time you modify an existing class, you can introduce a bug in the working code. The idea of this principle is to leave the working code untouched, so it does not get accidentally broken.

Let's have an example where this principle is not followed. We have the following existing and working code:

```
public interface Shape {
}

public class RectangleShape implements Shape {
  private int width;
  private int height;

  public RectangleShape(final int width, final int height) {
    this.width = width;
    this.height = height;
  }

  public int getWidth() {
    return width;
  }

  public int getHeight() {
```

```
    return height;
  }

  public void setWidth(final int newWidth) {
    width = newWidth;
  }

  public void setHeight(final int newHeight) {
    height = newHeight;
  }
}
```

Suppose we get an assignment to introduce support for square shapes. Let's try to modify the existing `RectangleShape` class, because a square is also a rectangle:

```
public class RectangleShape implements Shape {
  private int width;
  private int height;

  // Rectangle constructor
  public RectangleShape(final int width, final int height)  {
    this.width = width;
    this.height = height;
  }

  // Square constructor
  public RectangleShape(final int sideLength) {
    this.width = sideLength;
    this.height = sideLength;
  }

  public int getWidth() {
      return width;
  }

  public int getHeight() {
      return height;
  }

  public void setWidth(final int newWidth) {
    if (height == width) {
      //noinspection SuspiciousNameCombination
      height = newWidth;
    }

    width = newWidth;
  }

  public void setHeight(final int newHeight) {
    if (height == width) {
      //noinspection SuspiciousNameCombination
      width = newHeight;
    }

    height = newHeight;
  }
}
```

We needed to add a new constructor and modify two methods in the class. Everything works okay when we run tests. But we have introduced a subtle bug in the code: If we create a rectangle with an

equal height and width, the rectangle becomes a square, which is probably not what is wanted. This is a bug that can be hard to find in unit tests. This example showed that modifying an existing class can be problematic. We modified an existing class and accidentally broke it.

A better solution to introduce support for square shapes is to use the *open-closed principle* and create a new class that implements the `Shape` interface. Then we don't have to modify any existing class, and there is no risk of accidentally breaking something in the existing code. Below is the new `SquareShape` class:

```
public class SquareShape implements Shape {
  private int sideLength;

  public SquareShape(final int sideLength) {
    this.sideLength = sideLength;
  }

  public int getSideLength() {
    return sideLength;
  }

  public void setSideLength(final int newSideLength) {
    sideLength = newSideLength;
  }
}
```

An existing class can be safely modified by adding a new method in the following cases:

- The added method is a pure function, i.e., it always returns the same value for the same arguments and does not have side effects, i.e., it does not modify the object's state.
- The added method is read-only and tread-safe, i.e., it does not modify the object's state and accesses the object's state in a thread-safe manner in the case of multithreaded code. An example of a read-only method in a shape class would be a method that calculates the shape's area.
- Class is immutable, i.e., the added method (or any other method) cannot modify the object's state

There are a couple of cases where the modification of existing code is needed. One example is factories. When you introduce a new class, you need to modify the related factory to be able to create an instance of that new class. For example, if we had a `ShapeFactory` class, we would need to modify it to support the creation of `SquareShape` objects. Factories are discussed later in this chapter.

Another case is adding a new enum constant. You typically need to modify existing code to handle the new enum constant. If you forget to add the handling of the new enum constant somewhere in the existing code, typically, a bug will arise. For this reason, you should always safeguard switch-case statements with a *default* case that throws and if/else-if structures with an else branch that throws. You can also enable your static code analysis tool to report an issue if a switch statement's default case is missing or an else-branch is missing from an if/else-if structure. Also, some static code analysis tools can report an issue if you miss handling an enum constant in a switch-case

statement.

Here is an example of safeguarding an if/else-if structure in Java:

```
public enum FilterType {
  INCLUDE,
  EXCLUDE
}

interface Filter {
  boolean isFilteredOut(...);
}

public class FilterImpl implements Filter {
  private final FilterType filterType;

  public FilterImpl(final FilterType filterType, ...) {
    this.filterType = filterType;
    // ...
  }

  public boolean isFilteredOut(...) {
    if (filterType == FilterType.INCLUDE) {
      // ...
    } else if (filterType == FilterType.EXCLUDE) {
      // ..
    } else {
      // Safeguarding
      throw new IllegalArgumentException("Invalid filter type");
    }
  }
}
```

In TypeScript, safeguarding might be needed for union types also:

```
type FilterType = 'include' | 'exclude';

if (filterType === 'include') {
  // ...
} else if (filterType === 'include') {
  // ...
} else {
  // Safeguarding
  throw new Error("Invalid filter type");
}
```

We can notice from the above examples that if/else-if structures could be avoided with a better object-oriented design. For instance, we could create a `Filter` interface and two separate classes, `IncludeFilter` and `ExcludeFilter`, that implement the `Filter` interface. Using object-oriented design allows us to eliminate the `FilterType` enum and the if/else-if structure. This is known as the *replace conditionals with polymorphism* refactoring technique. Refactoring is discussed more in the next chapter. Below is the above Java example refactored to be more object-oriented:

```
interface Filter {
  boolean isFilteredOut(...);
```

```
}

public class IncludeFilter implements Filter {

  // ...

  public boolean isFilteredOut(...) {
    // ...
  }
}

public class ExcludeFilter implements Filter {

  // ...

  public boolean isFilteredOut(...) {
    // ...
  }
}
```

Liskov's Substitution Principle

> *Objects of a superclass should be replaceable with objects of its subclasses without breaking the application. I.e., objects of subclasses behave the same way as the objects of the superclass.*

Following *Liskov's substitution principle* guarantees semantic interoperability of types in a type hierarchy.

Let's have an example with a `RectangleShape` class and a derived `SquareShape` class:

```
public interface Shape {
  void draw();
}

public class RectangleShape implements Shape {
  private int width;
  private int height;

  public RectangleShape(final int width, final int height) {
    this.width = width;
    this.height = height;
  }

  public void draw() {
    // ...
  }

  public int getWidth() {
    return this.width;
  }

  public int getHeight() {
    return this.height;
  }

  public void setWidth(final int newWidth) {
```

```
    width = newWidth;
  }

  public void setHeight(final int newHeight) {
    height = newHeight;
  }
}

public class SquareShape extends RectangleShape {
  public SquareShape(final int sideLength) {
    super(sideLength, sideLength);
  }

  @Override
  public void setWidth(final int newWidth) {
    super.setWidth(newWidth);
    //noinspection SuspiciousNameCombination
    super.setHeight(newWidth);
  }

  @Override
  public void setHeight(final int newHeight) {
    //noinspection SuspiciousNameCombination
    super.setWidth(newHeight);
    super.setHeight(newHeight);
  }
}
```

The above example does not follow Liskov's substitution principle because you cannot set a square's width and height separately. This means that a square is not a rectangle from an object-oriented point of view. Of course, mathematically, a square is a rectangle. But when considering the above public API of the `RectangleShape` class, we can conclude that a square is not a rectangle because a square cannot fully implement the API of the `RectangleShape` class. We cannot substitute a square object for a rectangle object. What we need to do is to implement the `SquareShape` class without deriving from the `RectangleShape` class:

```
public class SquareShape implements Shape {
  private int sideLength;

  public SquareShape(final int sideLength) {
    this.sideLength = sideLength;
  }

  public void draw() {
    // ...
  }

  public int getSideLength() {
    return this.sideLength;
  }

  public void setSideLength(final int newSideLength) {
    sideLength = newSideLength;
  }
}
```

Liskov's substitution principle requires the following:

- A subclass must implement the superclass API and retain (or, in some cases, replace) the functionality of the superclass.
- A superclass should not have protected fields because it allows subclasses to modify the state of the superclass, which can lead to incorrect behavior in the superclass

Below is an example where a subclass extends the behavior of a superclass in the `doSomething` method. The functionality of the superclass is retained in the subclass making a subclass object substitutable for a superclass object.

```
public class SuperClass {
  // ...

  public void doSomething() {
    // ...
  }
}

public class SubClass extends SuperClass {
  // ...

  @Override
  public void doSomething() {
    super.doSomething();

    // Some additional behaviour...
  }
}
```

Let's have a concrete example of using the above strategy. We have the following `CircleShape` class defined:

```
public interface Shape {
  void draw();
}

public class CircleShape implements Shape {
  public void draw() {
    // draw the circle stroke
  }
}
```

Next, we introduce a class for filled circles:

```
public class FilledCircleShape extends CircleShape {
  @Override
  public void draw() {
     super.draw(); // draws the circle stroke
     // Fill the circle
  }
}
```

The `FilledCircleShape` class fulfills the requirements of Liskov's substitution principle. We can use an instance of the `FilledCircleShape` class everywhere where an instance of the

`CircleShape` class is wanted. The `FilledCircleShape` class does all that the `CircleShape` class does, plus adds some behavior (= filling the circle).

You can also completely replace the superclass functionality in a subclass:

```
public class ReverseArrayList<T> extends ArrayList<T>
{
  @Override
  public Iterator<T> iterator() {
    return new ReverseListIterator<>(this);
  }
}
```

The above subclass implements the superclass API and retains its behavior: The `iterator` method still returns an iterator. It just returns a different iterator compared to the superclass.

Interface Segregation and Multiple Inheritance Principle

> *Segregate a larger interface to micro interfaces with a single capability/behavior and construct larger interfaces by inheriting multiple micro interfaces.*

Let's have an example with several automobile classes:

```
public interface Automobile {
  void drive(Location start, Location destination);
  void carryCargo(double volumeInCubicMeters, double weightInKgs);
}

public class PassengerCar implements Automobile {
  // Implement drive and carryCargo
}

public class Van implements Automobile {
  // Implement drive and carryCargo
}

public class Truck implements Automobile {
  // Implement drive and carryCargo
}

public interface ExcavatingAutomobile extends Automobile {
  void excavate(...);
}

public class Excavator implements ExcavatingAutomobile {
  // Implement drive, carryCargo and excavate
}
```

Notice how the `Automobile` interface has two methods declared. This can limit our software if we later want to introduce other vehicles that could be just driven but unable to carry cargo. In an early phase, we should segregate two micro interfaces from the `Automobile` interface. A micro interface defines a single capability or behavior. After segregation, we will have the following two micro

interfaces:

```
public interface Drivable {
  void drive(Location start, Location destination);
}

public interface CargoCarriable {
  void carryCargo(double volumeInCubicMeters, double weightInKgs);
}
```

Now that we have two interfaces, we can use these interfaces also separately in our codebase. For example, we can have a list of drivable objects or a list of objects that can carry cargo. We still want to have an interface for automobiles, though. We can use *interface multiple inheritance* to redefine the `Automobile` interface to extend the two micro interfaces:

```
public interface Automobile extends Drivable, CargoCarriable {
}
```

If we look at the `ExcavatingAutomobile` interface, we can notice that it extends the `Automobile` interface and adds excavating behavior. Once again, we have a problem if we want to have an excavating machine that is not auto-mobile. The excavating behavior should be segregated into its own micro interface:

```
public interface Excavating {
  void excavate(...);
}
```

We can once again use the interface multiple inheritance to redefine the `ExcavatingAutomobile` interface as follows:

```
public interface ExcavatingAutomobile
                   extends Excavating, Automobile {
}
```

The `ExcavatingAutomobile` interface now extends three micro interfaces: `Excavating`, `Drivable`, and `CargoCarriable`. Where-ever you need an excavating, drivable, or cargo carriable object in your codebase, you can use an instance of the `Excavator` class there.

Let's have another example with a generic collection interface with TypeScript. We should be able to traverse a collection and also be able to compare two collections for equality. First, we define a generic `MyIterator` interface for iterators. It has two methods, as described below:

```
interface MyIterator<T> {
  hasNextElement(): boolean;
  getNextElement(): T;
}
```

Next, we can define the collection interface:

```
interface Collection<T> {
  createIterator(): MyIterator<T>;
  equals(anotherCollection: Collection<T>): boolean;
}
```

`Collection<T>` is an interface with two unrelated methods. Let's segregate those methods into two micro interfaces: `MyIterable` and `Equatable`. The `MyIterable` interface is for objects that you can iterate over. It has one method for creating new iterators. The `Equatable` interface's `equals` method is more generic than the `equals` method in the `Collection<T>` interface. You can equate an `Equatable<T>` object with another object of type `T`:

```
interface MyIterable<T> {
  createIterator(): MyIterator<T>;
}

interface Equatable<T> {
  equals(anotherObject: T): boolean;
}
```

We can use interface multiple inheritance to redefine the `Collection<T>` interface as follows:

```
interface Collection<T> extends MyIterable<T>,
                                Equatable<Collection<T>> {
}
```

We can implement the `equals` method by iterating elements in two collections and checking if the elements are equal:

```
abstract class AbstractCollection<T> implements Collection<T> {
  abstract createIterator(): MyIterator<T>;

  equals(anotherCollection: Collection<T>): boolean {
    const iterator = this.createIterator();
    const anotherIterator = anotherCollection.createIterator();
    let collectionsAreEqual =
      this.areEqual(iterator, anotherIterator);

    if (anotherIterator.hasNextElement()) {
      collectionsAreEqual = false;
    }

    return collectionsAreEqual;
  }

  private areEqual(
    iterator: MyIterator<T>,
    anotherIterator: MyIterator<T>
  ): boolean {
    while (iterator.hasNextElement()) {
      if (anotherIterator.hasNextElement()) {
        if (iterator.getNextElement() !==
            anotherIterator.getNextElement()) {
         return false;
        }
      } else {
        return false;
```

```
      }
    }

    return true;
  }
}
```

Collections can also be compared. Let's introduce support for such collections. First, we define a generic `Comparable<T>` interface for comparing an object with another object:

```
type ComparisonResult = 'isLessThan' | 'areEqual' | 'isGreaterThan' | 'unspecified';

interface Comparable<T> {
  compareTo(anotherObject: T): ComparisonResult;
}
```

Now we can introduce a comparable collection interface that allows comparing two collections of the same type:

```
interface ComparableCollection<T>
            extends Comparable<Collection<T>>, Collection<T> {
}
```

Let's define a generic sorting algorithm for collections whose elements are comparable:

```
function sort<T, U extends Comparable<T>, V extends Collection<U>>(
  collection: V
): V {
  // ...
}
```

Let's create two interfaces, `Inserting<T>` and `InsertingIterable<T>` for classes whose instances elements can be inserted into:

```
interface Inserting<T> {
  insert(element: T): void;
}

interface InsertingIterable<T> extends Inserting<T>,
                                       MyIterable<T> {
}
```

Let's redefine the `Collection` interface to extend the `InsertingIterable` interface because a collection is iterable, and you can insert elements into a collection.

```
interface Collection<T> extends InsertingIterable<T> {
}
```

Next, we introduce two generic algorithms for collections: `map` and `filter`. We can realize that those algorithms work with more abstract objects than collections. We benefit from interface segregation because instead of the `Collection<T>` interface, we can use the `MyIterable<T>`

and `InsertingIterable<T>` interfaces to create generic `map` and `filter` algorithms. Later it is possible to introduce some additional non-collection iterable objects that can utilize the algorithms as well. Below is the implementation of the `map` and `filter` functions:

```
function map<T, U>(
  source: MyIterable<T>,
  mapped: (sourceElement: T) => U,
  destination: InsertingIterable<U>
): InsertingIterable<U> {
  const sourceIterator = source.createIterator();

  while(sourceIterator.hasNextElement()) {
    const sourceElement = sourceIterator.getNextElement();
    destination.insert(mapped(sourceElement));
  }

  return destination;
}

function filter<T>(
  source: MyIterable<T>,
  isIncluded: (sourceElement: T) => boolean,
  destination: InsertingIterable<T>
): InsertingIterable<T> {
  const sourceIterator = source.createIterator();

  while (sourceIterator.hasNextElement()) {
    const sourceElement = sourceIterator.getNextElement();

    if (isIncluded(sourceElement)) {
      destination.insert(sourceElement);
    }
  }

  return destination;
}
```

Let's define the following concrete collection classes:

```
class List<T> implements Collection<T> {
  constructor(...args: T[]) {
    // ...
  }

  // ...
}

class Stack<T> implements Collection<T> {
  // ...
}

class MySet<T> implements Collection<T> {
  // ...
}
```

Now we can use the `map` and `filter` algorithms with the above-defined collection classes:

```
const numbers = new List<number>(1, 2, 3, 3, 3, 50, 60);
```

```
const isLessThan10 = (nbr: number) => nbr < 10;

const uniqueLessThan10Numbers =
  filter(numbers, isLessThan10, new MySet());

const doubled = (nbr: number) => 2 * nbr;
const stackOfDoubledNumbers = map(numbers, doubled, new Stack());
```

Let's create asynchronous version of the `map` algorithms:

```
interface MaybeCloseable {
  tryClose(): Promise<void>;
}

interface MaybeInserting<T> {
  tryInsert(value: T): Promise<void>;
}

interface MaybeCloseableInserting<T>
            extends MaybeCloseable, MaybeInserting<T> {
}

class MapError extends Error {
  // ...
}

async function tryMap<T, U>(
  source: MyIterable<T>,
  mapped: (sourceElement: T) => U,
  destination: MaybeCloseableInserting<U>
): Promise<void> {
  const sourceIterator = source.createIterator();

  try {
    while (sourceIterator.hasNextElement()) {
      const sourceElement = sourceIterator.getNextElement();
      await destination.tryInsert(mapped(sourceElement));
    }

    await destination.tryClose();
  } catch (error: any) {
    throw new MapError(error.message);
  }
}
```

Let's create a `FileLineInserter` class that implements the `MaybeCloseableInserting` interface:

```
const fs = require('fs');

class FileLineInserter<T extends { toString(): string }>
        implements MaybeCloseableInserting<T> {
  private writeStream: FS.WriteStream;

  constructor(private readonly filePathName: string) {
    this.writeStream =
      fs.createWriteStream(this.filePathName, { flags: 'a' });
  }

  async tryInsert(value: T): Promise<void> {
```

```
    try {
      const writePromise = new Promise((resolve, reject) => {
        const line = value.toString() + '\n';

        this.writeStream.write(line, (error: any) => {
          if (error) {
            reject(error);
          } else {
            resolve(undefined);
          }
        });
      });

      await writePromise;
    } catch (error: any) {
      throw new Error(error.message);
    }
  }

  tryClose(): Promise<void> {
    this.writeStream.close();
    return Promise.resolve();
  }
}
```

Let's use the above-defined `tryMap` algorithm and the `FileLineInserter` class to write doubled numbers (one number per line) to a file named *file.txt*:

```
const numbers = new List<number>(1, 2, 3, 2, 1, 50, 60);
const doubled = (nbr: number) => 2 * nbr;

try {
  await tryMap(numbers, doubled, new FileLineInserter('file.txt'));
} catch(error: any) { // error will be always MapError type.
  console.log(error.message);
}
```

Program Against Interfaces Principle (Generalized Dependency Inversion Principle)

> *Do not write programs where internal dependencies are concrete object types—instead, program against interfaces. An exception to this rule is data classes with no behavior (not counting simple getters/setters).*

An interface is used to define an abstract base type. Various implementations can be introduced that implement the interface. When you want to change the behavior of a program, you create a new class that implements an interface and then use an instance of that class. In this way, you can practice the *open-closed principle*. You can think of this principle as a prerequisite for using the *open-closed principle* effectively. The *program against interfaces principle* is a generalization of the *dependency inversion principle* from the SOLID principles:

> *The dependency inversion principle is a methodology for loosely coupling software classes.*

When following the principle, the conventional dependency relationships from high-level classes to low-level classes are reversed, thus making the high-level classes independent of the low-level implementation details.

The *dependency inversion principle* states:

- High-level classes should not import anything from low-level classes
- Abstractions (= interfaces) should not depend on concrete implementations (classes)
- Concrete implementations (classes) should depend on abstractions (= interfaces)

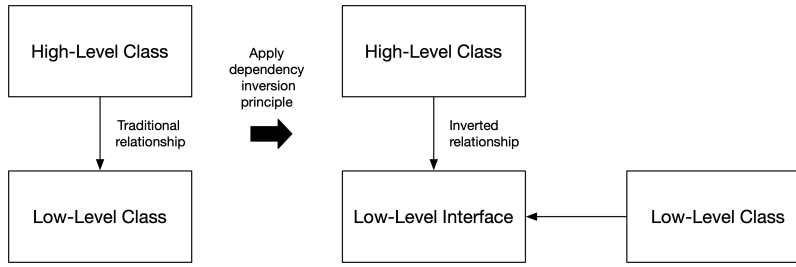

Fig 3.1 Dependency Inversion Principle

An interface is always an abstract type and cannot be instantiated. Below is an example of an interface:

```
public interface Shape {
  void draw();
  double calculateArea();
}
```

The name of an interface describes something abstract, which you cannot create an object of. In the above example, `Shape` is clearly something abstract. You cannot create an instance of `Shape` and then draw it or calculate its area because you don't know what shape it is. But when a class implements an interface, a concrete object of the class representing the interface can be created. Below is an example of three different classes that implement the `Shape` interface:

```
public class CircleShape implements Shape {
  private final int radius;

  public CircleShape(final int radius) {
    this.radius = radius;
  }

  public void draw() {
    // ...
  }

  public double calculateArea() {
```

```
    return Math.PI * radius * radius;
  }
}

public class RectangleShape implements Shape {
  private final int width;
  private final int height;

  public RectangleShape(final int width, final int height) {
    this.width = width;
    this.height = height;
  }

  public void draw() {
    // ...
  }

  public double calculateArea() {
    return width * height;
  }
}

public class SquareShape extends RectangleShape {
  public SquareShape(final int sideLength) {
   super(sideLength, sideLength);
  }
}
```

When using shapes in code, we should program against the `Shape` interface. In the below example, we make a high-level class `Canvas` dependent on the `Shape` interface, not on any of the low-level classes (`CircleShape`, `RectangleShape` or `SquareShape`). Now both the high-level `Canvas` class and all the low-level shape classes depend on abstraction only, the `Shape` interface. We can also notice that the high-level class `Canvas` does not import anything from the low-level classes. Also, the abstraction `Shape` does not depend on concrete implementations (classes).

```
public class Canvas {
  private final List<Shape> shapes = new ArrayList<>(10);

  public Canvas() {
  }

  public void add(final Shape shape) {
    shapes.add(shape);
  }

  public void drawShapes() {
    for(final var shape : shapes) {
      shape.draw();
    }
  }
}
```

A `Canvas` object can contain any shape and draw any shape. It can handle any of the currently defined concrete shapes and any new shape defined in the future.

If you did not program against an interface and did not use the dependency inversion principle,

your `Canvas` class would look like the following:

```
public class Circle {
  public void draw() {
    // ...
  }
}

public class Rectangle {
  public void draw() {
    // ...
  }
}

public class Square {
  public void draw() {
    // ...
  }
}

public class Canvas {
  private final List<Circle> circles = new ArrayList<>(10);
  private final List<Rectangle> rectangles = new ArrayList<>(10);
  private final List<Square> squares = new ArrayList<>(10);

  public Canvas() {
  }

  public void addCircle(final Circle circle) {
    circles.add(circle);
  }

  public void addRectangle(final Rectangle rectangle) {
    rectangles.add(rectangle);
  }

  public void addSquare(final Square square) {
    squares.add(square);
  }

  public void drawShapes() {
    for(final var circle : circles) {
      circle.draw();
    }

    for(final var rectangle : rectangles) {
      rectangle.draw();
    }

    for(final var square : squares) {
      square.draw();
    }
  }
}
```

The above high-level `Canvas` class is coupled with all the low-level classes (`Circle`, `Rectangle`, and `Square`). The `Canvas` class must be modified if a new shape type is needed. If something changes in the public API of any low-level class, the `Canvas` class needs to be modified accordingly.

Let's have another example. If you have read books or articles about object-oriented design, you may have encountered something similar as is presented in the below example:

```
public class Dog {
  public void walk() {
    // ...
  }

  public void bark() {
    // ...
  }
}

public class Fish {
  public void swim() {
    // ...
  }
}

public class Bird {
  public void fly() {
    // ...
  }

  public void sing() {
    // ...
  }
}
```

Three concrete implementations are defined above, but no interface is defined. Let's say we are making a game that has different animals. The first thing to do when coding the game is to remember to program against interfaces and thus introduce an `Animal` interface that we can use as an abstract base type. Let's try to create the `Animal` interface based on the above concrete implementations:

```
public interface Animal {
  void walk();
  void bark();
  void swim();
  void fly();
  void sing();
}

public class Dog implements Animal {
  public void walk() {
    // ...
  }

  public void bark() {
    // ...
  }

  public void swim() {
    throw new RuntimeException("Illegal operation");
  }

  public void fly() {
    throw new RuntimeException("Illegal operation");
```

```
  }

  public void sing() {
    throw new RuntimeException("Illegal operation");
  }
}
```

The above approach is wrong. We declare that the `Dog` class implements the `Animal` interface, but it does not do that. It implements only methods `walk` and `bark` while other methods throw an exception. We should be able to substitute any concrete animal implementation where an animal is required. But it is impossible because if we have a `Dog` object, we cannot safely call `swim`, `fly`, or `sing` methods because they will always throw.

The problem is that we defined the concrete classes before defining the interface. That approach is wrong. We should specify the interface first and then the concrete implementations. What we did above was the other way around.

When defining an interface, we should remember that we are defining an abstract base type, so we must think in abstract terms. We must consider what we want the animals to do in the game. If we look at the methods `walk`, `fly`, and `swim`, they are all concrete actions. But what is the abstract action common to these three concrete actions? It is *move*. And walking, flying, and swimming are all ways of moving. Similarly, if we look at the `bark` and `sing` methods, they are also concrete actions. What is the abstract action common to these two concrete actions? It is *makeSound*. And barking and singing are both ways to make a sound. If we use these abstract actions, our `Animal` interface looks like the following:

```
public interface Animal {
  void move();
  void makeSound();
}
```

We can now redefine the animal classes to implement the new `Animal` interface:

```
public class Dog implements Animal {
  public void move() {
    // walk
  }

  public void makeSound() {
    // bark
  }
}

public class Fish implements Animal {
  public void move() {
    // swim
  }

  public void makeSound() {
    // Intentionally no operation
    // (Fishes typically don't make sounds)
  }
```

```
}

public class Bird implements Animal {
  public void move() {
    // fly
  }

  public void makeSound() {
    // sing
  }
}
```

Now we have a correct object-oriented design and can program against the `Animal` interface. We can call the `move` method when we want an animal to move and the `makeSound` method when we want an animal to make a sound.

After realizing that some birds don't fly at all, we can easily enhance our design. We can introduce two different implementations:

```
public abstract class AbstractBird implements Animal {
  public abstract void move();

  public void makeSound() {
    // sing
  }
}

public class FlyingBird extends AbstractBird {
  public void move() {
    // fly
  }
}

public class NonFlyingBird extends AbstractBird {
  public void move() {
    // walk
  }
}
```

We might also later realize that not all birds sing but make different sounds. Ducks quack, for example. Instead of using inheritance as was done above, an even better alternative is to use *object composition*. We compose the `Bird` class of behavioral classes for moving and making sounds:

```
public interface Mover {
  void move();
}

public interface SoundMaker {
  void makeSound();
}

public class Bird implements Animal {
  private final Mover mover;
  private final SoundMaker soundMaker;

  public Bird(
    final Mover mover,
```

```
    final SoundMaker soundMaker
  ) {
    this.mover = mover;
    this.soundMaker = soundMaker;
  }

  public void move() {
    mover.move();
  }

  public void makeSound() {
    soundMaker.makeSound();
  }
}
```

Now we can create birds with various behaviors for moving and making sounds. We can use the *factory pattern* to create different birds. The *factory pattern* is described in more detail later in this chapter. Let's introduce three different moving and sound-making behaviors and a factory to make three kinds of birds: goldfinches, ostriches, and domestic ducks.

```
public class Flyer implements Mover {
  public void move() {
    // fly
  }
}

public class Runner implements Mover {
  public void move() {
    // run
  }
}

public class Walker implements Mover {
  public void move() {
    // walk
  }
}

public class GoldfinchSoundMaker implements SoundMaker {
  public void makeSound() {
    // Sing goldfinch specific songs
  }
}

public class OstrichSoundMaker implements SoundMaker {
  public void makeSound() {
    // Make ostrich specific sounds like whistles,
    // hoots, hisses, growls, and deep booming growls
    // that sound like the roar of a lion
  }
}

public class Quacker implements SoundMaker {
  public void makeSound() {
    // quack
  }
}

public enum BirdType {
  GOLDFINCH,
```

```
  OSTRICH,
  DOMESTIC_DUCK
}

public class BirdFactory {
  public Bird createBird(final BirdType birdType) {
    return switch(birdType) {
      case GOLDFINCH ->
        new Bird(new Flyer(),
                 new GoldfinchSoundMaker());

      case OSTRICH ->
        new Bird(new Runner(),
                 new OstrichSoundMaker());

      case DOMESTIC_DUCK ->
        new Bird(new Walker(),
                 new Quacker());

      default ->
        throw new IllegalArgumentException("Unsupported type");
    };
  }
}
```

Clean Microservice Design Principle

> *The clean microservice design promotes object-oriented design with separation of concerns achieved by dividing software into layers using the dependency inversion principle (programming against interfaces).*

Clean microservice design comes with the following benefits:

- Not tied to any single framework
- Not tied to any single API technology like REST or GraphQL
- Unit testable
- Not tied to a specific client (works with web, desktop, console, and mobile clients)
- Not tied to a specific database
- Not dependent on any specific external service implementation

A clean API microservice design consists of the following layers:

- Controller, Interface adapters
- Use cases
- (Business) Entities

Uses cases and entities together form the *model* of the service, also called the *business logic*.

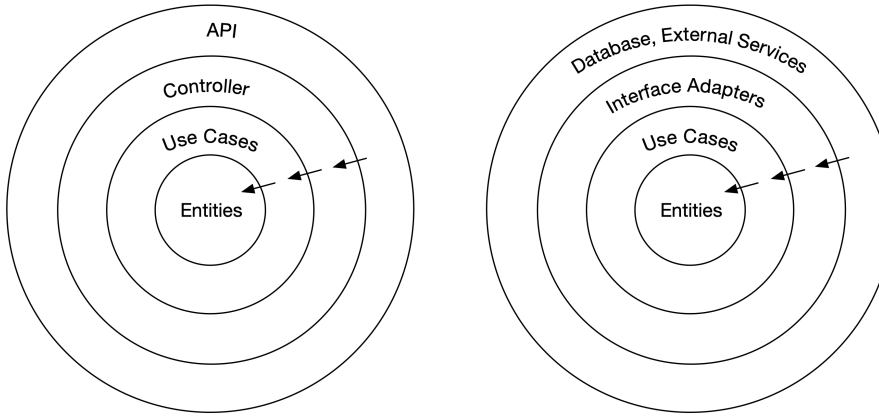

Fig 3.2 Clean Microservice Design

The direction of dependencies in the above diagrams is shown with arrows. We can see that the microservice API depends on the controller we create. The controller depends on the use cases. The use case layer depends on entities. The purpose of the use case layer is to orchestrate operations on the entities.

Let's have a real-life example of creating an API microservice called *order-service*, which handles orders in an e-commerce software system. First, we define a REST API controller using Java and Spring Boot:

```
@RestController
@RequestMapping("/orders")
public class RestOrderController {
  @Autowired
  private OrderService orderService;

  @PostMapping
  @ResponseStatus(HttpStatus.CREATED)
  public final Order createOrder(
    @RequestBody final OrderArg orderArg
  ) {
    return orderService.createOrder(orderArg);
  }

  // Other API methods...
}
```

The API offered by the microservice depends on the controller, as seen in the above diagram. The API is currently a REST API, but we could create and use a GraphQL controller. Then our API, which depends on the controller, is a GraphQL API. Below is a partial implementation of a GraphQL controller with Java and Spring Boot:

```
@Controller
public class GraphQlOrderController {
  @Autowired
  private OrderService orderService;
```

```
  @MutationMapping
  public final Order createOrder(
    @Argument final OrderArg orderArg
  ) {
    return orderService.createOrder(orderArg);
  }

  // Other API methods...
}
```

The *`RestOrderController`* and `GraphQlOrderController` classes depend on the `OrderService` interface, which is part of the use case layer. Notice that the controllers do not rely on a concrete implementation of the use cases but depend on an interface according to the *dependency inversion principle*. Below is the definition for the `OrderService` interface:

```
public interface OrderService {
  Order createOrder(OrderArg orderArg);
  Order getOrderById(Long id);
  Iterable<Order> getOrderByUserAccountId(Long userAccountId);
  void updateOrder(Long id, OrderArg orderArg);
  void deleteOrderById(Long id);
}
```

The below `OrderServiceImpl` class implements the `OrderService` interface:

```
@Service
public class OrderServiceImpl implements OrderService {
  private static final String ORDER = "Order";

  @Autowired
  private OrderRepository orderRepository;

  @Override
  public final Order createOrder(
    final OrderArg orderArg
  ) {
    final var order = Order.from(orderArg);
    return orderRepository.save(order);
  }

  @Override
  public final Order getOrderById(final Long id) {
    return orderRepository.findById(id)
              .orElseThrow(() ->
                new EntityNotFoundError(ORDER, id));
  }

  // Rest of the methods...
}
```

The `OrderServiceImpl` class has a dependency on an order repository. This dependency is also inverted. The `OrderServiceImpl` class depends only on the `OrderRepository` interface. The order repository is used to orchestrate the persistence of order entities. Note that there is not any direct dependency on a database.

Below is the `OrderRepository` interface:

```
public interface OrderRepository {
  Order save(Order order);
  Order findById(Long id);
  // ...
}
```

The `OrderRepository` interface depends only on the `Order` entity class. You can introduce a class called an *interface adapter* that implements the `OrderRepository` interface. A database interface adapter adapts a particular concrete database to the `OrderRepository` interface. Entity classes do not depend on anything except other entities to create hierarchical entities. For example, the `Order` entity consists of `OrderItem` entities.

When implementing a clean microservice design, everything is wired together using configuration and dependency injection. For example, an instance implementing the `OrderRepository` interface is created according to configuration and injected into an `OrderServiceImpl` instance by the Spring framework. In the case of Spring, the dependency injector is configured using a configuration file and annotations. The configuration file can be used to configure what database is used. Additionally, the Spring dependency injector creates an instance of the `OrderServiceImpl` class and injects it where an `OrderService` object is wanted.

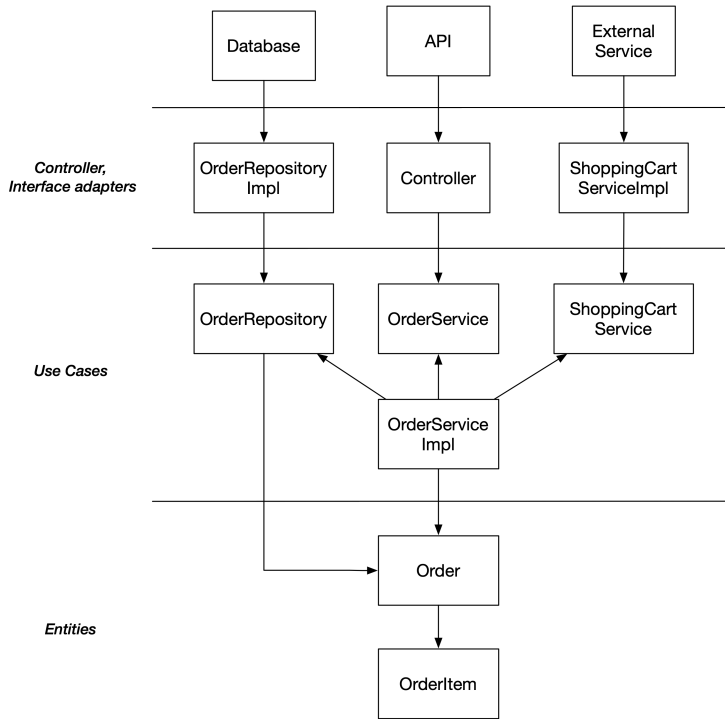

Fig 3.3 Clean Microservice Design for Order Service

The dependency injector is the only place in a microservice that contains references to concrete implementations. In many frameworks, the dependency injector is not a visible component, but its usage is configured using a configuration file and annotations. For example, in Spring, the `@Autowired` annotation tells the dependency injector to inject a concrete implementation into the annotated class field or constructor parameter. The *dependency injection principle* is discussed more in a later section of this chapter. The dependency inversion principle and dependency injection principle usually go hand in hand. Dependency injection is used for wiring interface dependencies so that those become dependencies on concrete implementations, as seen in the figure below.

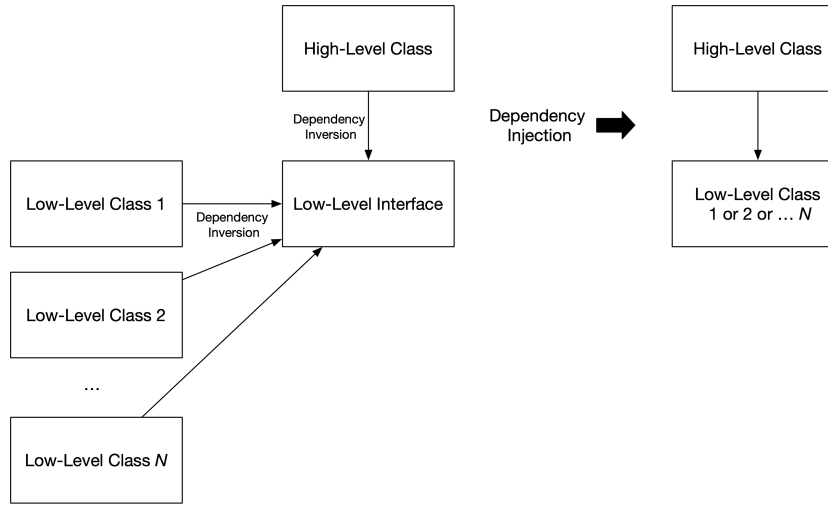

Fig 3.4 Dependency Injection

Let's add a feature where the shopping cart is emptied when an order is created:

```
@Service
public class OrderServiceImpl implements OrderService {
  @Autowired
  private OrderRepository orderRepository;

  @Autowired
  private ShoppingCartService shoppingCartService;

  @Override
  public final Order createOrder(
    final OrderArg orderArg
  ) {
    final var order = Order.from(orderArg);
    final var savedOrder = orderRepository.save(order);
    shoppingCartService.emptyCart(order.userAccountId);
    return savedOrder;
  }
}
```

As you can see from the above code, the `OrderServiceImpl` class is not depending on any concrete implementation of the shopping cart service. We can create an *interface adapter* class that is a concrete implementation of the `ShoppingCartService` interface. That interface adapter class connects to a particular external shopping cart service, for example, via REST API. Once again, the dependency injector will inject a concrete `ShoppingCartService` implementation to an instance of the `OrderServiceImpl` class.

Note that the above `createOrder` method is not production quality because it lacks a transaction.

Uniform Naming Principle

> *Use a uniform way to name interfaces, classes, and functions.*

This section presents conventions for uniformly naming interfaces, classes, and functions.

Naming Interfaces and Classes

> *Classes represent a thing or an actor. They should be named consistently so that the class name ends with a noun. An interface represents an abstract thing, actor, or capability. Interfaces representing a thing or an actor should be named like classes but using an abstract noun. Interfaces representing a capability should be named according to the capability.*

When an interface represents an abstract thing, name it according to that abstract thing. For example, if you have a drawing application with various geometrical objects, name the geometrical object interface `Shape`. It is a simple abstract noun. Names should always be the shortest, most descriptive ones. There is no reason to name the geometrical object interface as `GeometricalObject` or `GeometricalShape`, if we can use simply `Shape`.

When an interface represents an abstract actor, name it according to that abstract actor. The name of an interface should be derived from the functionality it provides. For example, if there is a `parseConfig` method in the interface, the interface should be named `ConfigParser`, and if an interface has a `validateObject` method, the interface should be named `ObjectValidator`. Don't use mismatching name combinations like a `ConfigReader` interface with a `parseConfig` method or an `ObjectValidator` interface with a `validateData` method.

When an interface represents a capability, name it according to that capability. Capability is something that a concrete class is capable of doing. For example, a class could be sortable, iterable, comparable, equitable, etc. Name the respective interfaces according to the capability: `Sortable`, `Iterable`, `Comparable`, and `Equitable`. The name of an interface representing a capability

usually ends with *able* or *ing*.

Don't name interfaces starting with an *I*. Instead, use an *Impl* postfix for class names to distinguish a class from an interface when needed. You should be programming against interfaces, and if every interface has its name prefixed with *I*, it just adds unnecessary noise to the code. Use the *I* prefix only if it is a programming language convention.

Some examples of class names representing a thing are: `Account`, `Order`, `RectangleShape`, and `CircleShape`. In a class inheritance hierarchy, the names of classes usually refine the interface name or the base class name. For example, if there is an `InputMessage` interface, then there can be different concrete implementations (= classes) of the `InputMessage` interface. They can represent an input message from different sources, like `KafkaInputMessage` and `HttpInputMessage`. And there could be different subclasses for different data formats: `AvroBinaryKafkaInputMessage` or `JsonHttpInputMessage`.

The interface or base class name should be retained in the class or subclass name. Class names should follow the pattern: <*class-purpose*> + <*interface-name*> or <*sub-class-purpose*> + <*super-class-name*>, e.g., `Kafka + InputMessage = KafkaInputMessage` and `AvroBinary + KafkaInputMessage = AvroBinaryKafkaInputMessage`. Name abstract classes with the prefix `Abstract`.

Don't add a design pattern name to a class name if it does not bring any real benefit. For example, suppose we have a `DataStore` interface, a `DataStoreImpl` class, and a class that is wrapping a `DataStore` instance and uses the *proxy pattern* to add caching functionality to the wrapped data store. We should not name the caching class `CachingProxyDataStore` or `CachingDataStoreProxy` . The word *proxy* does not add significant value, so the class should be named simply `CachingDataStore`.

Naming Functions

> *Functions should do one thing, and the name of a function should describe what the function does. The function name must contain a verb that indicates what the function does. The function name should usually start with a verb, but exceptions exist. If a function returns a value, try to name the function so that the function name describes what it returns.*

The general rule is to name a function so that the purpose of the function is clear. A good function name should not make you think.

Below is an example of an interface containing two methods named with simple verbs only. It is not necessary to name the methods as `startThread` and `stopThread` because the methods are already part of the `Thread` interface, and it is self-evident what the `start` method starts and what the `end` method ends.

```
public interface Thread {
  void start();
  void stop();
}
```

Let's have another Java example:

```
grpcChannel.shutdown().awaitTermination(30, TimeUnit.SECONDS);
```

The above example has two issues with the `shutdown` function. Most people probably assume that calling the `shutdown` function will shut down the channel and return after the channel is shut down without any return value. But now the `shutdown` function is returning something. It is not necessarily self-evident what it returns. But we can notice that the `shutdown` function does not wait for the channel termination.

It would be better to rename the `shutdown` function as `requestShutdown` because it better describes what the function does. Also, we should name the `awaitTermination` to `awaitShutdown` because we should not use two different terms *shutdown* and *termination* to denote a single thing.

```
final var shutdownPromise = grpcChannel.requestShutdown();
shutdownPromise.awaitShutdown(30, TimeUnit.SECONDS);
```

Let's have an example in JavaScript:

```
fetch(url).then(response => response.json()).then(...);
```

In the above example, we have the following issue: the `fetch` function does not properly describe what it does. According to the documentation, it fetches a resource. But it does not return a resource. It returns a response object. Whenever possible, the function name should indicate what the function returns. The `fetch` performs an action on a resource and does not always return a resource. The action is specified by giving an HTTP verb as a parameter to the function (GET is the default HTTP verb). The most common actions are GET, POST, PUT and DELETE. If you issue a PUT request for a REST API, you don't usually get the resource back. The same is, of course, valid for a DELETE request. You cannot get the resource back because it was just deleted.

We could name the function `performActionOnResource`, but that is a pretty long name and does not communicate the return value type. We should name the `fetch` function `makeHttpRequest` (or `sendHttpRequest`) to indicate that it is making an HTTP request. The new function name also communicates that it returns an HTTP response. Another possibility is introducing an actor class with static methods for different HTTP methods, for example: `HttpClient.makeGetRequest(url)`.

In the above example, the `json` function name is missing a verb. It should contain the verb *parse* because that is what it is doing. The function name should also tell what it parses: the response

body. We should also add a *try* prefix to indicate that the function can throw (more about the *try* prefix and error handling in general in the next chapter). Below is the example with renamed functions:

```
makeHttpRequest(url).then(response =>
  response.tryParseBodyJson()).then(...);
```

Many languages offer streams that can be written to, like the standard output stream. Streams are usually buffered, and the actual writing to the stream does not happen immediately. For example, the below statement does not necessarily write to the standard output stream immediately. It buffers the text to be written later when the buffer is flushed to the stream. This can happen when the buffer is full, when some time has elapsed since the last flush or when the stream is closed.

```
stdOutStream.write(...);
```

The above statement is misleading and could be corrected by renaming the function to describe what it actually does:

```
stdOutStream.writeOnFlush(...);
```

The above function name immediately tells a developer that writing happens only on flush, and the developer can consult the function documentation to determine when the flushing happens. You can introduce a convenience method to perform a write with an immediate flush:

```
// Instead of this:
stdOutStream.writeOnFlush(...);
stdOutStream.flush();

// User can do this:
stdOutStream.writeWithFlush(...);
```

Many times function's action is associated with a target, for example:

```
public interface ConfigParser {
  Configuration tryParseConfig(...);
}
```

When a function's action has a target, it is useful to name the function using the following pattern: <*action-verb*> + <*action-target*>, for example, `parse` + `config` = `parseConfig`.

We can drop the action target from the function name if the function's first parameter describes the action target. It is not wrong to keep the action target in the function name, though. But if it can be dropped, it usually makes the function call statements read better. In the below example, the word "*config*" appears repeated: `tryParseConfig(configJson)`, which makes the function call statement read a bit clumsy.

```
final var configuration = configParser.tryParseConfig(configJson);
```

We can drop the action target from the function name:

```
public interface ConfigParser {
  Configuration tryParse(final String configJson);
}
```

As shown below, this change makes the code read better, presuming we use a descriptive variable name. And we should, of course, always use a descriptive variable name.

```
final var configuration = configParser.tryParse(configJson);
```

Here is another example:

```
public class Vector<T> {
  void pushBack(final T value); // OK
  void pushBackValue(final T value); // Not ideal,
                                     // word "value" repeated
}
```

Let's imagine we have the following function:

```
public class KafkaAdminClient {
  void create(final String topic);
}
```

The above function name should be used only when a topic is the only thing a Kafka admin client can create. We cannot call the above function in the following way:

```
kafkaAdminClient.create("xyz");
```

We need to introduce a properly named variable:

```
final var topic = "xyz";
kafkaAdminClient.create(topic);
```

In languages where you can use named function parameters, the following is possible:

```
// Python
kafkaAdminClient.create(topic = "xyz");

// Swift
kafkaAdminClient.create(topic: "xyz");
```

Preposition in Function Name

Use a preposition in a function name when needed to clarify the function's purpose.

You don't need to add a preposition to a function name if the preposition can be assumed (i.e., the preposition is implicit). In many cases, only one preposition can be assumed. If you have a function named `wait`, the preposition `for` can be assumed, and if you have a function named `subscribe`,

the preposition `to` can be assumed. We don't need to use function names `waitFor` and `subscribeTo`.

Suppose a function is named `laugh(person: Person)`. Now we have to add a preposition because none can be assumed. We should name the function either `laughWith(person: Person)` or `laughAt(person: Person)`.

The following two sections present examples of better naming some existing functions in programming languages.

Example 1: Renaming JavaScript Array Methods

Adding elements to a JavaScript array is done with the `push` method. Where does it push the elements? The method name does not say anything. There are three possibilities:

1. At the beginning
2. Somewhere in the middle
3. At the end

Most definitely, it is not the second one, but it still leaves two possibilities. Most people correctly guess that it pushes elements to the end. To make it 100% clear where the elements are pushed, this function should be named `pushBack`. Then it does not make anybody think where the elements are pushed. Remember that a good function name does not make you think.

Popping an element from an array is done with the `pop` method. But where does it pop from? If you read the method description, it tells that the element is popped at the back. To make it 100% clear, this method should be named `popBack`.

The `Array` class also contains methods `shift` and `unshift`. They are like `push` and `pop` but operate at the beginning of an array. Those method names are extremely non-descriptive and should be named `popFront` and `pushFront`.

There are several methods in the JavaScript `Array` class for finding elements in an array. Here is the list of those methods:

- `find` (finds the first element where the given predicate is true)
- `findIndex` (find the index of the first element where the given predicate is true)
- `includes` (returns true or false based on if the given element is found in the array)
- `indexOf` (returns the first index where the given element is found)
- `lastIndexOf` (returns the last index where the given element is found)

Here are the suggested new names for the above functions:

- find ==> findFirstWhere
- findIndex ==> findFirstIndexWhere
- includes ==> include
- indexOf ==> findFirstIndexOf
- lastIndexOf ==> findLastIndexOf

Below are examples of these new function names in use:

```
const numbers = [1, 2, 3, 4, 5, 5];
const numberIsEven = nbr => (nbr % 2) === 0;
const firstEvenNumber = numbers.findFirstWhere(numberIsEven);
const firstEvenNumberIndex = numbers.findFirstIndexWhere(numberIsEven);
const numbersIncludeFour = numbers.include(4);
const firstIndexOfFive = numbers.findFirstIndexOf(5);
const lastIndexOfFive = numbers.findLastIndexOf(5);
```

Example 2: Renaming C++ Casting Expressions

C++ contains several casting expressions. They are not functions per se, but they act and look a lot like functions. Below is a list of C++ cast operations for which I am going to present some alternative names:

- const_cast
- reinterpret_cast
- static_cast
- dynamic_cast

The first cast, `const_cast`, performs possibly multiple things like adding/removing const-ness and/or volatility. The `const_cast` is often used to add or remove a single `const`. It would be better to define separate operations for all the different cases: `remove_const`, `add_const`, `add_volatile`, and `remove_volatile`.

The second cast, `reinterpret_cast`, converts between types by reinterpreting the underlying bit pattern. What it does is that it forces the cast and could be named `force_cast`.

The third cast, `static_cast`, is a standard cast performed during the compilation phase. A compilation error will be given if the cast cannot be performed. We could name this cast with an even simpler name: `cast`.

The fourth cast, `dynamic_cast`, does two different things: upcasting and down-casting in the class inheritance hierarchy. We could name these operations: `try_cast_to_base_class` and `try_cast_to_derived_class`. A cast to a base class is always successful if the given argument

is an object of the derived class. I prefixed the operations with *try* to tell that they can throw. Alternative operations `cast_to_base_class` and `cast_to_derived_class` that return optional values could be introduced.

Naming Method Pairs

Methods in a class can come in pairs. A typical example is a pair of getter and setter methods. When you define a method pair in a class, name the methods logically. The methods in a method pair often do two opposite things, like getting or setting a value. If you are unsure how to name one of the methods, try to find an antonym for a word. For example, if you have a method whose name starts with "create" and are unsure how to name the method for the opposite action, try a Google search: "create antonym".

Here is a non-comprehensive list of some method names that come in pairs:

- get/set (getters and setters)
    - Name a boolean getter with the same name as the respective field, e.g., `boolean isDone()`
    - Name a boolean setter with *set* + <*boolean field name*>, e.g., `void setIsDone(boolean isDone)`
- get/put (especially when accessing a collection)
- read/write
- add/remove
- store/retrieve
- open/close
- load/save
- initialize/destroy
- create/destroy
- insert/delete
- start/stop
- pause/resume
- start/finish
- increase/decrease
- increment/decrement
- construct/destruct
- obtain/relinquish
- acquire/release
- reserve/release
- startup/shutdown

- login/logout
- begin/end
- launch/terminate
- publish/subscribe
- join/detach
- <something>/un<something>, e.g. assign/unassign, install/uninstall, subscribe/unsubscribe, follow/unfollow
- <something>/de<something>, e.g. serialize/deserialize, allocate/deallocate, encode/decode, encrypt/decrypt
- something/dis<something>, e.g. connect/disconnect, associate/disassociate

The `apt` tool in Debian/Ubuntu-based Linux has an `install` command to install a package, but the command for uninstalling a package is `remove`. It should be `uninstall`. The Kubernetes package manager Helm has this correct. It has an `install` command to install a Helm release and an `uninstall` command to uninstall it.

Naming Boolean Functions (Predicates)

> *The naming of boolean functions (predicates) should be such that when reading the function call statement, it reads as a boolean statement that can be true or false.*

In this section, we consider naming functions that are predicates and return a boolean value. Here I don't mean functions that return true or false based on the success of the executed action, but cases where the function call is used to evaluate a statement as true or false. The naming of boolean functions should be such that when reading the function call statement, it makes a statement that can be true or false. Below are some examples:

```
public class Response {
  public boolean hasError() {
    // ...
  }
}

public class String {
  public boolean isEmpty() {
    //...
  }

  public boolean startsWith(final String anotherString) {
    //...
  }

  public boolean endsWith(final String anotherString) {
    // ...
  }

  public boolean contains(final String anotherString) {
    // ...
```

```
  }
}

// Here we have a statement: response has error? true or false?
if (response.hasError()) {
  // ...
}

// Here we have a statement: line is empty? true or false?
final String line = fileReader.readLine();
if (line.isEmpty()) {
  // ...
}

// Here we have statement: line starts with a space character?
// true or false?
if (line.startsWith(" ")) {
    // ...
}

// Here we have statement: line ends with a semicolon?
// true or false?
if (line.endsWith(";")) {
    // ...
}

public class Thread {
  public boolean shouldTerminate() {
    // ...
  }

  public boolean isPaused() {
    // ...
  }

  public boolean canResumeExecution() {
    // ...
  }

  public void run() {
    // ...

    // Here we have statement: [this] should terminate?
    // true or false?
    if (shouldTerminate()) {
      return;
    }

    // Here we have statement: [this] is paused and
    // [this] can resume execution? true or false?
    if (isPaused() && canResumeExecution()) {
      // ...
    }

    // ...
  }
}
```

A boolean returning function is correctly named when you call the function in code and can read that function call statement in plain English. Below is an example of incorrect and correct naming:

```
public class Thread {
  public boolean stopped() { // Incorrect naming
    // ...
  }

  public boolean isStopped() { // Correct naming
    // ...
  }
}

if (thread.stopped()) {
  // Here we have: if thread stopped
  // This is not a statement with a true or false answer
  // It is a second conditional form,
  // asking what would happen if thread stopped.
  // ...
}

// Here we have statement: if thread is stopped
// true or false?
if (thread.isStopped()) {
  // ...
}
```

From the above examples, we can notice that many names of boolean-returning functions start with either *is* or *has* and follows the below pattern:

- is + <adjective>, e.g. isOpen, isRunning or isPaused
- has + <noun>

Also, these two forms can be relatively common:

- should + <verb>
- can + <verb>

But as we saw with the `startsWith, endsWith` and `contains` functions, a boolean returning function name can start with any verb in third-person singular form (i.e., ending with an *s*). If you have a collection class, its boolean method names should have a verb in the plural form, for example: `numbers.include(...)` instead of `numbers.includes(...)`. Name your collection variables always in plural form (e.g., `numbers` instead of `numberList`). We will discuss the uniform naming principles for variables in the next chapter.

Do not include the *does* word in a function name, like *doesStartWith*, *doesEndWith*, or *doesContain*. Adding the *does* word doesn't add any real value to the name, and such function names are awkward to read when used in code, for example:

```
final String line = textFileReader.readLine();

// "If line does start with" sound awkward
if (!line.doesStartWith(" ")) {
  // ...
```

```
}
```

When you want to use the past tense in a function name, use a *did* prefix in the function name, for example:

```
public class DatabaseOperation {
  public void execute() {
    // ...
  }

  // Method name not OK. This is a second conditional form
  // if (dbOperation.startedTransaction())...
  public boolean startedTransaction() {
    // ...
  }

  // Method name OK, no confusion possible
  public boolean didStartTransaction() {
    // ...
  }
}
```

Naming Builder Methods

A builder class is used to create builder objects that build a new object of a particular type. If you wanted to construct a URL, a `UrlBuilder` class could be used for that purpose. Builder class methods add properties to the built object. For this reason, it is recommended to name builder class methods starting with the verb *add*. The method that finally builds the wanted object should be named simply *build* or *build* + <*build-target*>, for example, `buildUrl`. I prefer the longer form to remind the reader what is being built. Below is an example of naming the methods in a builder class:

```
public class UrlBuilder {
  public UrlBuilder() {
    // ...
  }

  public UrlBuilder addScheme(final String scheme) {
   // ...
   return this;
  }

  public UrlBuilder addHost(final String host) {
    // ...
    return this;
  }

  public UrlBuilder addPort(final int port) {
    // ...
    return this;
  }

  public UrlBuilder addPath(final String path) {
    // ...
    return this;
  }
```

```
  public UrlBuilder addQuery(final String query) {
    // ...
    return this;
  }

  public Url buildUrl() {
    // ...
  }
};

final var url = new UrlBuilder()
  .addScheme("https://")
  .addHost("google.com")
  .buildUrl();
```

Naming Methods with Implicit Verbs

Factory method names usually start with the verb *create*. Factory methods can be named so that the *create* verb is implicit, for example:

```
Optional.of(final T value)
Optional.empty() // Not optimal, 'empty' can be confused as a verb
Either.withLeft(final L value)
Either.withRight(final L value)
SalesItem.from(final SalesItemArg salesItemArg)
```

The explicit versions of the above method names would be:

```
Optional.createOf(final T value)
Optional.createEmpty()
Either.createWithLeft(final L value)
Either.createWithRight(final L value)
SalesItem.createFrom(final SalesItemArg salesItemArg)
```

Similarly, conversion methods can be named so that the *convert* verb is implicit. Conversion methods without a verb usually start with the *to* preposition, for example:

```
value.toString();
object.toJson();
```

The explicitly named versions of the above methods would be:

```
value.convertToString();
object.convertToJson();
```

You can access a collection element in some languages using the method `at(index)`. Here the implicit verb is *get*. I recommend using method names with implicit verbs sparingly and only in circumstances where the implicit verb is self-evident and does not force a developer to think.

Naming Property Getter Functions

Property getter functions are usually named *get* + <*property-name*>. It is also possible to name a property getter that does not have a respective setter using just the property name. This is acceptable in cases where the property name cannot be confused with a verb. Below is an example of property getters:

```
final var list = new MyList();

list.size(); // OK
list.length(); // OK
list.empty(); // NOT OK, empty can be a verb.
list.isEmpty(); // OK
```

Naming Lifecycle Methods

Lifecycle methods are called on certain occasions only. Lifecycle method names should answer the question: When or "on what occasion" will this method be called? Examples of good names for lifecycle methods are: `onInit`, `onError`, `onSuccess`, `afterMount` and `beforeUnmount`. In React, there are lifecycle methods in class components called `componentDidMount`, `componentDidUpdate` and `componentWillUnmount`. There is no reason to repeat the class name in the lifecycle method names. Better names would have been: `afterMount`, `afterUpdate`, and `beforeUnmount`.

Naming Function Parameters

Naming rules for function parameters are mostly the same as for variables. *Uniform naming principle* for variables is described in the next chapter in more detail.

There are some exceptions, like naming object parameters. When a function parameter is an object, the name of the object class can be left out from the parameter name when the parameter name and the function name implicitly describe the class of the parameter. This exception is acceptable because the function parameter type can always be easily checked by looking at the function signature. And this should be easily done with a glance because a function should be short (a maximum of 5-7 statements). Below is an example of naming object type parameters:

```
// Word 'Location' repeated, not optimal, but allowed
drive(startLocation: Location, destinationLocation: Location): void

// Better way
// When we think about 'drive' and 'start' or 'destination',
// we can assume that 'start' and 'destination' mean locations
drive(start: Location, destination: Location): void
```

Some programming languages like Swift allow adding so-called *external names* to function parameters. Using external names can make a function call statement read better, as shown below:

```
func drive(from start: Location, to destination: Location) {
  // ...
}
```

```
func send(
  message: String,
  from sender: Person,
  to recipient: Person
) {
  // ...
}


let startLocation = new Location(...);
let destLocation = new Location(...);
drive(from: startLocation, to: destLocation);

let message = "Some message";
let person = new Person(...);
let anotherPerson = new Person(...);
send(message, from: person, to: anotherPerson);
```

Encapsulation Principle

> *A class should encapsulate its state so that access to the state happens only via public class methods.*

Encapsulation is achieved by declaring class fields private. You can create getter and setter methods if you need the state to be modifiable outside the class. However, encapsulation is best ensured if you don't need to create getter and setter methods for the class fields. Do not automatically implement getter and setter methods for every class. Only create those accessor methods if needed, like when the class represents a modifiable data structure. And only generate setter methods for class fields that need to be modified outside the class.

Immutable Objects

The best way to ensure the encapsulation of an object's state is to make the object immutable. This means that once the object is created, its state cannot be modified afterward. Immutability ensures you cannot accidentally or intentionally modify the object's state. Modifying the object's state outside the object can be a source of bugs.

When creating an immutable object, you give the needed properties for the object in the constructor, and after that, those properties cannot be modified. An immutable class has fields marked as `final`, `const`, or `readonly` (depending on the programming language), meaning no setter methods can be created for the class.

If you need to modify an immutable object, the only way is to create a new object with different values given to the constructor. The drawback of this approach is that a performance penalty is introduced when creating new objects as compared to modifying existing objects' properties only.

But in many cases, this penalty is negligible compared to the benefits of immutability. For example, strings are immutable in many languages, like Java and JavaScript. Once you create a string, you cannot modify it. You can only create new strings.

Immutability also requires that getters and other methods returning a value may not return a modifiable class field, like an array. If you returned such an array from a method, that array could be modified by adding or removing elements without the "owning" object being aware of that.

Don't Leak Modifiable Internal State Outside an Object Principle

Beware when you return values from your class methods. It is possible that a method accidentally returns some internal state of the object that can be modified later by the method caller. Returning modifiable state from a class method breaks the encapsulation. Static code analyzers for some languages, like Java, can warn you about situations where internal state leaks outside the object.

You can safely return primitive or so-called value types from class methods. Those types include types like Java boolean, int, and long. You can also safely return an immutable object, like a Java string. But you cannot safely return a mutable collection, for example.

There are two ways how to protect against leaking internal state outside an object:

1. Return a copy of the modifiable internal state
2. Return an unmodifiable version of the modifiable internal state

Regarding the first approach, when a copy is returned, the caller can use it as they like. Changes made to the copied object don't affect the original object. I am primarily talking about making a shallow copy. In many cases, a shallow copy is enough. For example, a list of primitive values, immutable strings, or immutable objects does not require a deep copy of the list. But you should make a deep copy when needed.

The copying approach can cause a performance penalty, but in many cases, that penalty is insignificant. In JavaScript, you can easily create a copy of an array:

```
const values = [1, 2, 3, 4, 5];
const copyOfValues = [...values];
```

The second approach requires you to create an unmodifiable version of a modifiable object and return that unmodifiable object. Some languages offer an easy way to create unmodifiable versions of certain objects. In Java, you can create an unmodifiable version of a `List`, `Map`, or `Set` using `Collections.unmodifiableList`, `Collections.unmodifiableMap`, or `Collections.unmodifiableSet` factory method, respectively.

You can also create an unmodifiable version of a class by yourself. Below is an example in Java:

```
public interface MyList<T> {
  void addToEnd(T item);
  Optional<T> getItem(int index);
}

public class UnmodifiableMyList<T> implements MyList<T> {
  private final MyList<T> list;

  public UnmodifiableMyList(final MyList<T> list) {
    this.list = list;
  }

  public void addToEnd(final T item) {
    throw new UnsupportedOperationException(...);
  }

  public Optional<T> getItem(final int index) {
    return list.getItem(index);
  }
}
```

In the above example, the unmodifiable list class takes another list (a modifiable list) as a constructor argument. It only implements the `MyList` interface methods that don't attempt to modify the wrapped list. In this case, it implements only the `getItem` method that delegates to the respective method in the `MyList` class. The `UnmodifiableMyList` class methods that attempt to modify the wrapped list should throw an error. The `UnmodifiableMyList` class utilizes the *proxy pattern* by wrapping an object of the `MyList` class and partially allowing access to the `MyList` class methods.

In C++, you can return an unmodifiable version by declaring the return type as `const`, for example:

```
std::shared_ptr<const std::vector<std::string>>
getStringValues() const;
```

Now callers of the `getStringValues` method cannot modify the returned vector of strings because it is declared const.

Unmodifiable and immutable objects are slightly different. No one can modify an immutable object, but when you return an unmodifiable object from a class method, that object can still be modified by the owning class, and modifications are visible to everyone that has received an unmodifiable version of the object. If this is something undesirable, you should use a copy instead.

Don't Assign From a Method Parameter to a Modifiable Field

If a class receives modifiable objects as constructor or method arguments, it is typically best practice not to assign those arguments to the internal state directly. If they are assigned directly, the class can, on purpose or accidentally modify those argument objects, which is probably not what the constructor or method caller expects.

There are two ways to handle this situation:

1. Store a copy of the modifiable argument object to the class's internal state
2. Store an unmodifiable version of the modifiable argument object to the class's internal state

Below is an example of the second approach:

```
public class MyClass {
  private final List<Integer> values;

  public MyClass(final List<Integer> values) {
    this.values = Collections.unmodifiableList(values);
  }
}
```

Real-life Example of Encapsulation Violation: React Class Component's State

React class component's state is not properly encapsulated. React documentation instructs that the `state` property should be modified directly using `this.state` in a `Component` subclass constructor. For example:

```
import { Component } from 'react';

class ButtonClickCounter extends Component {
  constructor(props) {
    super(props);

    this.state = {
      clickCount: 0
    };
  }
}
```

It is not good object-oriented design that the `state` property is public or protected in the `Component` class. You should not modify the base class's `state` property in the `ButtonClickCounter` subclass. The proper way to initialize the state in an object-oriented manner would be to give the initial state as a parameter to the `Component` class constructor using `super`. However, the following is not supported by React:

```
import { Component } from 'react';

export default class ButtonClickCounter extends Component {
  constructor(props) {
    // This is not possible in real life
    super(props, {
      clickCount: 0
    });
  }
}
```

Setting the state is done with the `setState` method defined in the `Component` class, but accessing the state happens directly through the `state` property. This leads to a problem where you cannot use `this.state` when calling the `setState` method because that can lead to erroneous behavior, according to the React documentation. So the following is not allowed:

```
incrementClickCount = () =>
  this.setState({
    clickCount: this.state.clickCount + 1
  });
```

Below is an example of using the `setState` method correctly in a React class component:

```
import { Component } from 'react';

export default class ButtonClickCounter extends Component {
  constructor(props) {
    super(props);

    this.state = {
      clickCount: 0
    };
  }

  incrementClickCount = () =>
    this.setState(({ clickCount }) => ({
      clickCount: clickCount + 1
    }));

  render() {
    return (
      <>
        Click count: {this.state.clickCount}
        <button onClick={this.incrementClickCount} />
      </>
    );
  }
}
```

Accessing the state in the `Component` subclasses should be done using a getter `getState`, not directly accessing the `state` property. Below is the above example modified to use the *imaginary* `getState` method:

```
import { Component, Fragment } from 'react';

export default class ButtonClickCounter extends Component {
  constructor(props) {
    super(props, {
      clickCount: 0
    });
  }

  incrementClickCount = () =>
    this.setState({
      clickCount: this.getState().clickCount + 1
    });
```

```
  render() {
    return (
      <>
        Click count: {this.getState().clickCount}
        <button onClick={this.incrementClickCount} />
      </>
    );
  }
}
```

Object Composition Principle

> *In object-oriented design, like in real life, objects are constructed by constructing larger objects from smaller objects. This is called object composition. Prefer object composition over inheritance.*

For example, a car object can be composed of an engine and transmission object (to name a few). Objects are rarely "composed" by deriving from another object, i.e., using inheritance. But first, let's try to specify classes that implement the below `Car` interface using inheritance:

```
public interface Car {
  void drive(
    Location start,
    Location destination
 );
}

public class CombustionEngineCar implements Car {
  public void drive(
    final Location start,
    final Location destination
  ) {
    // ...
  }
}

public class ElectricEngineCar implements Car {
  public void drive(
    final Location start,
    final Location destination
  ) {
    // ...
  }
}

public class ManualTransmissionCombustionEngineCar
         extends CombustionEngineCar {
  public void drive(
    final Location start,
    final Location destination
  ) {
    // ...
  }
}
```

```
public class AutomaticTransmissionCombustionEngineCar
         extends CombustionEngineCar {
  public void drive(
    final Location start,
    final Location destination
  ) {
    // ...
  }
}
```

If we wanted to add other components to a car, like a two or four-wheel drive, the number of classes needed would increase by three. If we wanted to add a design property (sedan, hatchback, wagon, or SUV) to a car, the number of needed classes would explode, and the class names would become ridiculously long. We can notice that inheritance is not the correct way to build more complex classes.

Class inheritance creates an *is-a* relationship between a superclass and its subclasses. Object composition creates a *has-a* relationship. We can claim that `ManualTransmissionCombustionEngineCar` *is a* kind of `CombustionEngineCar`, so basically, we are not doing anything wrong here, one might think. But when designing classes, you should first determine if object composition could be used: is there a *has-a* relationship? Can you declare a class as a property of another class? If the answer is yes, then composition should be used instead of inheritance.

All the above things related to a car are actually properties of a car. A car *has an* engine. A car *has a* transmission. It *has a* two or four-wheel drive and design. We can turn the inheritance-based solution into a composition-based solution:

```
public interface Drivable {
  void drive(
    Location start,
    Location destination
  );
}

public interface Engine {
  // Methods like start, stop ...
}

public class CombustionEngine implements Engine {
  // Methods like start, stop ...
}

public class ElectricEngine implements Engine {
  // Methods like start, stop ...
}

public interface Transmission {
  // Methods like changeGear ...
}

public class AutomaticTransmission implements Transmission {
  // Methods like changeGear ...
}
```

```
public class ManualTransmission implements Transmission {
  // Methods like changeGear ...
}

// Define DriveType here...
// Define Design here...

public class Car implements Drivable {
  private final Engine engine;
  private final Transmission transmission;
  private final DriveType driveType;
  private final Design design;

  public Car(
    final Engine engine,
    final Transmission transmission,
    final DriveType driveType,
    final Design design
  ) {
    this.engine = engine;
    this.transmission = transmission;
    this.driveType = driveType;
    this.design = design;
  }

  public void drive(
    final Location start,
    final Location destination
  ) {
    // To implement functionality, delegate to
    // component classes, for example:

    // engine.start();
    // transmission.shiftGear(...);
    // ...
    // engine.stop();
  }
}
```

Let's have a more realistic example with different chart types in TypeScript. At first, this sounds like a case where inheritance could be used: We have some abstract base charts that different concrete charts extend, for example:

```
interface Chart {
  renderView(): JSX.Element;
  updateData(...): void;
}

abstract class AbstractChart implements Chart {
  abstract renderView(): JSX.Element;
  abstract updateData(...): void;

  // Implement some common functionality
  // shared by all chart types
}

abstract class XAxisChart extends AbstractChart {
  abstract renderView(): JSX.Element;

  updateData(...): void {
    // This is common for all x-axis charts,
```

```
    // like ColumnChart, LineChart and AreaChart
  }
}

class ColumnChart extends XAxisChart {
  renderView(): JSX.Element {
    // ...

    return (
      <XYZChart
        type="column"
        data={data}
        options={options}...
      />;
    );
  }
}

// LineChart class definition here...
// AreaChart class definition here...

abstract class NonAxisChart extends AbstractChart {
  abstract renderView(): JSX.Element;

  updateData(...): void {
    // This is common for all non-x-axis charts,
    // like PieChart and DonutChart
  }
}

class PieChart extends NonAxisChart {
  renderView(): JSX.Element {
    // ...

    return (
      <XYZChart
        type="pie"
        data={data}
        options={options}...
      />;
    );
  }
}

class DonutChart extends PieChart {
  renderView(): JSX.Element {
    // ...

    return (
      <XYZChart
        type="donut"
        data={data}
        options={options}...
      />;
    );
  }
}
```

The above class hierarchy looks manageable: there should not be too many subclasses that need to be defined. We can, of course, think of new chart types, like a geographical map or data table for which we could add subclasses. One problem with a deep class hierarchy arises when you need to

change or correct something related to a particular chart type. Let's say you want to change or correct some behavior related to a pie chart. You will first check the `PieChart` class if the behavior is defined there. If you can't find what you are looking for, you need to navigate to the base class of the `PieChart` class (`NonAxisChart`) and look there. And you might need to continue this navigation until you reach the base class where the behavior you want to change or correct is located. Of course, if you are incredibly familiar with the codebase, you might be able to locate the correct subclass on the first try. But in general, this is not a straightforward task.

Using class inheritance can introduce class hierarchies where some classes have significantly more methods than other classes. For example, in the chart inheritance chain, the `AbstractChart` class probably has significantly more methods than classes at the end of the inheritance chain. This class size difference creates an imbalance between classes making it hard to reason about what functionality each class provides.

Even if the above class hierarchy might look okay at first sight, currently, there lies one problem. We have hardcoded what kind of chart view we are rendering. We are using the *XYZ* chart library and rendering `XYZChart` views. Let's say we would like to introduce another chart library called *ABC*. We want to use both chart libraries in parallel so that the open-source version of our data visualization application uses the *XYZ* chart library, which is open source. The paid version of our application uses the commercial *ABC* chart library. When using class inheritance, we must create new classes for each concrete chart type for the *ABC* chart library. So, we would have two classes for each concrete chart type, like here for the pie chart:

```
class XYZPieChart extends XyzNonAxisChart {
  renderView(): JSX.Element {
    // ...

    return (
      <XYZChart
        type="pie"
        data={data}
        options={options}...
      />;
    );
  }
}

class ABCPieChart extends AbcNonAxisChart {
  renderView(): JSX.Element {
    // ...

    return (
      <ABCPieChart
        dataSeries={dataSeries}
        chartOptions={chartOptions}...
      />;
    );
  }
}
```

Implementing the above functionality using composition instead of inheritance has several

benefits:

- It is more apparent what behavior each class contains
- There is no significant size imbalance between classes, where some classes are huge and others relatively small
- You can split chart behaviors into classes as you find fit, and is in accordance with the *single responsibility principle*

In the below example, we have split some chart behavior into two types of classes: chart view renderers and chart data factories:

```
interface Chart {
  renderView(): JSX.Element;
  updateData(...): void;
}

interface ChartViewRenderer {
  renderView(data: ChartData, options: ChartOptions): JSX.Element;
}

interface ChartDataFactory {
  createData(...): ChartData
}

// ChartData...
// ChartOptions...

class ChartImpl implements Chart {
  private data: ChartData;
  private options: ChartOptions;

  constructor(
    private readonly viewRenderer: ChartViewRenderer,
    private readonly dataFactory: ChartDataFactory
  ) {
    // ...
  }

  renderView(): JSX.Element {
    return this.viewRenderer.renderView(this.data, this.options);
  }

  updateData(...): void {
    this.data = this.dataFactory.createData(...);
  }
}

class XYZPieChartViewRenderer implements ChartViewRenderer {
  renderView(data: ChartData, options: ChartOptions): JSX.Element {
    // ...

    return (
      <XYZPieChart
        data={dataInXyzChartLibFormat}
        options={optionsInXyzChartLibFormat}...
      />;
    );
```

```
  }
}

class ABCPieChartViewRenderer implements ChartViewRenderer {
   renderView(data: ChartData, options: ChartOptions): JSX.Element {
     // ...

     return (
       <ABCPieChart
         dataSeries={dataInAbcChartLibFormat}
         chartOptions={optionsInAbcChartLibFormat}...
       />;
     );
   }
}

// ABCColumnChartViewRenderer...
// XYZColumnChartViewRenderer...

type ChartType = 'column' | 'pie';

interface ChartFactory {
  createChart(chartType: ChartType): Chart;
}

class ABCChartFactory implements ChartFactory {
  createChart(chartType: ChartType): Chart {
    switch(chartType) {
      case 'column':
        return new ChartImpl(new ABCColumnChartViewRenderer,
                             new XAxisChartDataFactory());
      case 'pie':
        return new ChartImpl(new ABCPieChartViewRenderer,
                             new NonAxisChartDataFactory());

      default:
        throw new Error('Invalid chart type');
    }
  }
}

class XYZChartFactory implements ChartFactory {
  createChart(chartType: ChartType): Chart {
    switch(chartType) {
      case 'column':
        return new ChartImpl(new XYZColumnChartViewRenderer,
                             new XAxisChartDataFactory());
      case 'pie':
        return new ChartImpl(new XYZPieChartViewRenderer,
                             new NonAxisChartDataFactory());

      default:
        throw new Error('Invalid chart type');
    }
  }
}
```

The `XYZPieChartViewRenderer` and `ABCPieChartViewRenderer` classes use the *adapter pattern* as they convert the supplied data and options to an implementation (ABC or XYZ chart library) specific interface.

We can easily add more functionality by composing the `ChartImpl` class of more classes. There could be, for example, a title formatter, tooltip formatter class, y/x-axis label formatter, and event handler classes.

```
class ChartImpl implements Chart {
  private data: ChartData;
  private options: ChartOptions;

  constructor(
    private readonly viewRenderer: ChartViewRenderer,
    private readonly dataFactory: ChartDataFactory,
    private readonly titleFormatter: ChartTitleFormatter,
    private readonly tooltipFormatter: ChartTooltipFormatter,
    private readonly xAxisLabelFormatter: ChartXAxisLabelFormatter,
    private readonly eventHandler: ChartEventHandler
  ) {
    // ...
  }

  renderView(): JSX.Element {
    return this.viewRenderer.renderView(this.data, this.options);
  }

  updateData(...): void {
    this.data = this.dataFactory.createData(...);
  }
}

class ABCChartFactory implements ChartFactory {
  createChart(chartType: ChartType): Chart {
    switch(chartType) {
      case 'column':
        return new ChartImpl(new ABCColumnChartViewRenderer,
                             new XAxisChartDataFactory(),
                             new ChartTitleFormatterImpl(),
                             new XAxisChartTooltipFormatter(),
                             new ChartXAxisLabelFormatterImpl(),
                             new ColumnChartEventHandler());

     case 'pie':
       return new ChartImpl(new ABCColumnChartViewRenderer,
                            new NonAxisChartDataFactory(),
                            new ChartTitleFormatterImpl(),
                            new NonAxisChartTooltipFormatter(),
                            new NullXAxisLabelFormatter(),
                            new NonAxisChartEventHandler());

      default:
        throw new Error('Invalid chart type');
    }
  }
}
```

Domain-Driven Design Principle

Domain-driven design (DDD) is a software design approach where software is modeled to

> *match the language of the problem domain that the software tries to solve. DDD is hierarchical. The top-level domain can be divided into subdomains which can be further divided into subdomains.*

DDD means that the structure of software, and the names appearing in the code (interface, class, function, and variable names) should match the domain. For example, in a banking software system, names like *Account*, *withdraw*, *deposit*, *makePayment* and *LoanApplication* should be used. The top-level domain of a software system should be divided into smaller subdomains. And each subdomain should be implemented as a separate application or software component. For example, a development team can be dedicated to the loan applications subdomain and another team to payments. Developers in a team need to know about their team's subdomain. And when interfacing with other domains, they need to know enough about the other domains to understand the interfaces. In this way, a single team will have a smaller set of concepts to comprehend and remember. Product managers and the chief architect should have a good grasp of the top-level domain, i.e., they should understand *the big picture*.

Domain-Driven Design Example 1: Data Exporter Microservice

Let's have a DDD example with a microservice for exporting data. Data exporting will be our top-level domain. The development team should participate in the DDD and object-oriented design (OOD) process. It is very likely that an expert-level software developer, e.g., the team tech lead, could do the DDD and OOD alone, but it is not how it should be done. Other team members, especially the junior ones, should be involved to learn and develop their skills further.

The DDD process is started by first defining the big picture (top-level domain) based on requirements from the product management and the architecture team:

> *Data exporter handles data that consists of messages that contain multiple fields. Data exporting should happen from an input system to an output system. During the export, various transformations to the data can be made, and the data formats in the input and output systems can differ.*

We can specify four subdomains for the data exporter microservice based on the requirements above:

- Consume messages from the input system and decode them into an internal message
- Internal message
- Transform internal messages
- Encode transformed messages to a wanted format and produce them to the output system

We can name the subdomains as shown in the below picture:

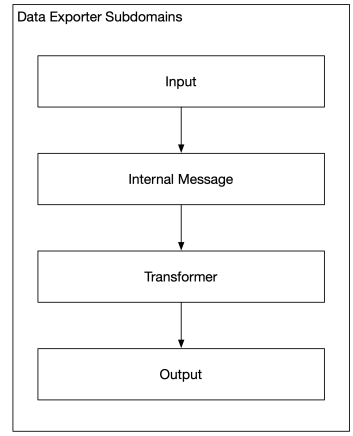

Figure 3.4 Data Exporter Subdomains

When considering the *Input* domain in more detail, we can figure out that it consists of the following subdomains, interfaces and classes:

- Input message
    - Contains the message consumed from the input data source
    - `InputMessage` is an interface that can have several concrete implementations, like `KafkaInputMessage` representing an input message consumed from a Kafka data source
- Input message consumer
    - Consumes messages from the input data source and creates `InputMessage` instances
    - `InputMessageConsumer` is an interface that can have several concrete implementations, like `KafkaInputMessageConsumer` for consuming messages from a Kafka data source
- Input Message decoder
    - Decodes input messages into internal messages
    - `InputMessageDecoder` is an interface that can have several concrete implementations, like `AvroBinaryInputMessageDecoder`, which decodes input messages encoded in Avro binary format
- Input configuration
    - Input configuration reader
        - Reads the domain's configuration
        - `InputConfigReader` is an interface that can have several concrete implementations, like `LocalFileSystemInputConfigReader` or

`HttpRemoteInputConfigReader`

- Input configuration parser
    - Parses the read configuration to produce an `InputConfig`
    - `InputConfigParser` is an interface that can have several concrete implementations, like `JsonInputConfigParser` or `YamlInputConfigParser`
- `InputConfig` instance contains parsed configuration for the domain, like the input data source type, host, port, and input data format.

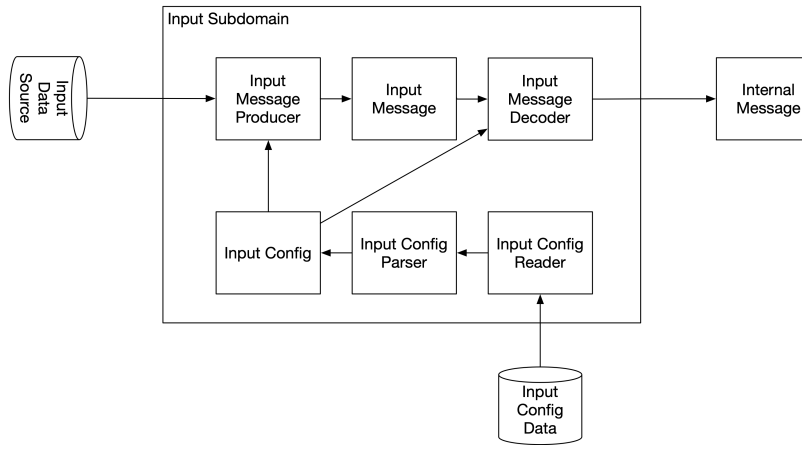

Figure 3.5 Input Subdomain

When considering the *Internal Message* domain in more detail, we can figure out that it consists of the following interfaces and classes:

- Internal Message
    - Internal message consists of one or more internal message fields
    - `InternalMessage` is an interface for a class that provides an internal representation of an input message
- Internal Message Field
    - `InternalMessageField` is an interface for classes representing a single field of an internal message

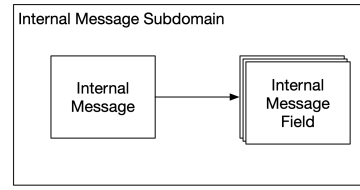

Figure 3.6 Internal Message Subdomain

When considering the *Transformer* domain in more detail, we can figure out that it consists of the following subdomains, interfaces, and classes:

- Field transformer
    - `FieldTransformers` is a collection of `FieldTransformer` objects
    - A Field transformer transforms the value of an input message field into a value of an output message field
    - `FieldTransformer` is an interface that can have several concrete implementations, like `FilterFieldTransforer`, `CopyFieldTransformer`, `TypeConversionFieldTransformer` and `ExpressionTransformer`
- Message Transformer
    - `MessageTransformer` takes an internal message and transforms it using field transformers
- Transformer configuration
    - Transformer configuration reader
        - Reads the domain's configuration
        - `TransformerConfigReader` is an interface that can have several concrete implementations, like `LocalFileSystemTransformerConfigReader`
    - Transformer configuration parser
        - Parses read configuration to produce a `TransformerConfig`
        - `TransformerConfigParser` is an interface that can have several concrete implementations, like `JsonTransformerConfigParser`
    - `TransformerConfig` instance contains parsed configuration for the *Transformer* domain

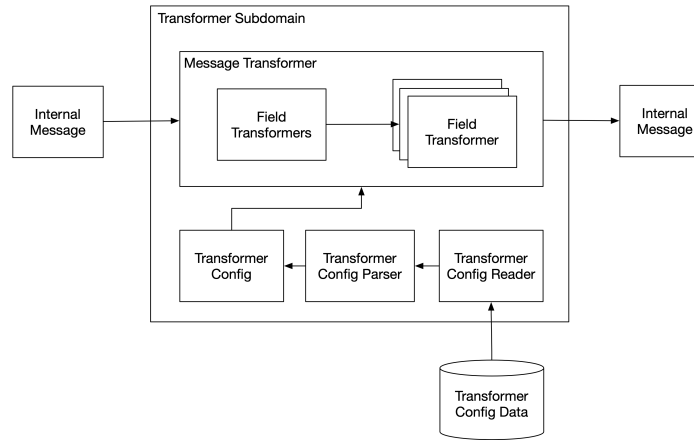

Figure 3.7 Transformer Subdomain

When considering the *Output* domain in more detail, we can figure out that it consists of the following subdomains, interfaces and classes:

- Output Message encoder
    - Encodes transformed message to an output message with a specific data format
    - `OutputMessageEncoder` is an interface that can have several concrete implementations, like `CsvOutputMessageEncoder`, `JsonOutputMessageEncoder`, `AvroBinaryOutputMessageEncoder`
- Output message
    - `OutputMessage` is an interface for container objects to hold output messages as a byte sequence
- Output message producer
    - Produces output messages to the output destination
    - `OutputMessageProducer` is an interface that can have several concrete implementations, like `KafkaMessageProducer`
- Output configuration
    - Output configuration reader
        - Reads the domain's configuration
        - `OutputConfigReader` is an interface that can have several concrete implementations, like `LocalFileSystemOutputConfigReader`
    - Output configuration parser
        - Parse the read configuration to an `OutputConfig`
        - `OutputConfigParser` is an interface that can have several concrete implementations, like JsonOutputConfigParser

- `OutputConfig` instance contains parsed configuration for the domain, like output destination type, host, port, and the output data format

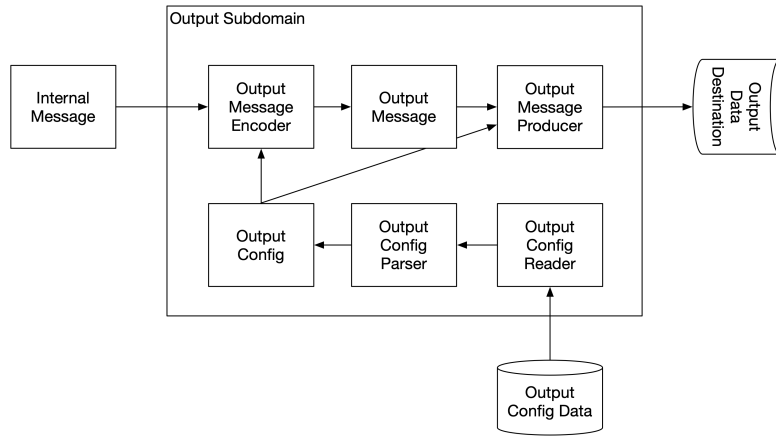

Figure 3.8 Output Subdomain

If you combine the above design diagrams, they form a data processing pipeline that can be implemented in the following way:

```
void DataExporterApp::run()
{
  while(m_isRunning)
  {
    const auto inputMessage =
      m_inputMessageConsumer.consumeInputMessage();

    const auto internalMessage =
      m_inputMessageDecoder.decodeToInternalMessage(inputMessage);

    const auto transformedMessage =
      m_messageTransformer.transform(*internalMessage);

    const auto outputMessage =
      m_outputMessageEncoder.encode(transformedMessage);

    m_outputMessageProducer.produce(outputMessage);
  }
}
```

And the `MessageTransformer::transform` method can be implemented in the following way:

```
std::unique_ptr<InternalMessage> MessageTrasformer::transform(
  const InternalMessage& internalMessage
)
{
  const auto transformedMessage =
```

```
    std::make_unique<InternalMessageImpl>();

  std::ranges::for_each(m_fieldTransformers,
                        [&internalMessage, &transformedMessage]
                        (const auto& fieldTransformer) {
    fieldTransformer.transform(internalMessage,
                               transformedMessage);
  });

  return transformedMessage;
}
```

Domain-Driven Design Example 2: Anomaly Detection Microservice

Let's have another DDD example with an anomaly detection microservice. The purpose of the microservice is to detect anomalies in measurement data. This concise description of the microservice's purpose reveals the two subdomains of the microservice:

- Anomaly
- Measurement

Let's first analyze the *Measurement* subdomain in more detail. We can identify the following subdomains for it:

- Measurement configuration
    - Loads and parses the following configuration
        - Measurement data sources
        - Measurements
- Measurement query
    - Represents a query for fetching measurement data
- Measurement data source
    - Represents a data source against which measurement queries can be executed
- Measurement data
    - Represents a result of an executed measurement query
    - Measurement data scaler
        - Scales the measurement data using a particular technique
- Measurement
    - Represents a measurement (contains properties like name, measurement query, etc.)

The *Anomaly* subdomain contains the following subdomains:

- Anomaly detection
    - Anomaly detection configuration
        - Loads and parses anomaly detection rules
    - Anomaly detection rule
        - Specifies how anomalies should be detected for a measurement
    - Anomaly detector
        - Detects anomalies in a measurement according to the anomaly detection rule using a trained anomaly model
    - Anomaly detection engine
        - A thread for detecting anomalies for the given anomaly detection configuration
- Anomaly model
    - Trained anomaly model
    - Anomaly model trainer
        - Trains an anomaly model using a specific AI technique, like self-organizing maps
    - Anomaly model training engine
        - A thread for training an anomaly model
- Anomaly indicator
    - Anomaly indicator representation
    - Anomaly indicator serializer
        - Serializes an anomaly indicator representation
        - Different serializers can be implemented, like JSON
    - Anomaly indicator publisher
        - Publishes a serialized anomaly indicator
        - Different publishers can be implemented, like Kafka or REST API

The two domains, anomaly and measurement, can be developed in parallel. The anomaly domain interfaces with the measurement domain to fetch data for a particular measurement from a particular data source. The development effort of both the anomaly and measurement domains can be further split to achieve even more development parallelization. For example, one developer could work with anomaly detection, another with anomaly model training, and the third with anomaly indicators.

Design Patterns

The following sections present 25 design patterns, most of which are made famous by the *Gang of Four* and their book *Design Patterns*. Design patterns are divided into creational, structural, and behavioral patterns.

Design Patterns for Creating Objects

This section describes design patterns for creating objects. The following design patterns will be presented:

- Factory pattern
- Abstract factory pattern
- Factory method pattern
- Builder pattern
- Singleton pattern
- Prototype pattern
- Object pool pattern

Factory Pattern

> *Factory pattern allows deferring what kind of object will be created to the point of calling the* ***create*** *method of the factory.*

A factory typically consists of precisely one or several methods for creating objects of a particular type.

Below is an example `ConfigParserFactory` that has a single `create` method for creating different kinds of `ConfigParser` objects. In the case of a single create method, the method usually contains a switch-case statement or an if/else-if structure. Factories are the only place where extensive switch-case statements or if/else-if structures are allowed in object-oriented programming. If you have a lengthy switch-case statement or long if/else-if structure somewhere else in code, that is typically a sign of a non-object-oriented design.

```
public interface ConfigParser {
  // ...
}

public class JsonConfigParser implements ConfigParser {
  // ...
}
```

```
public class YamlConfigParser implements ConfigParser {
  // ...
}

public enum ConfigFormat {
  JSON,
  YAML
}

public final class ConfigParserFactory {
  public static ConfigParser createConfigParser(
    final ConfigFormat configFormat
  ) {
    return switch(configFormat) {
      case JSON -> new JsonConfigParser();
      case YAML -> new YamlConfigParser();
      default ->
        throw new IllegalArgumentException(
          "Unsupported config format"
        );
    };
  }
}
```

Below is an example of a factory with multiple *create* methods:

```
public final class ShapeFactory {
  public static Shape createCircleShape(final int radius) {
    return new CircleShape(radius);
  }

  public static Shape createRectangleShape(
    final int width,
    final int height
  ) {
    return new RectangleShape(width, height);
  }

  public static Shape createSquareShape(final int sideLength) {
    return new SquareShape(sideLength);
  }
}
```

Abstract Factory Pattern

> *In the abstract factory pattern, there is an abstract factor (= factory interface) and one or more concrete factories (factory classes that implement the factory interface).*

The abstract factory pattern is an extension of the earlier described *factory pattern*. Usually, the abstract factory pattern should be used instead of the plain factory pattern. Below is an example of an abstract `ConfigParserFactory` with one concrete implementation:

```
public interface ConfigParserFactory {
  ConfigParser createConfigParser(ConfigFormat configFormat);
}

public class ConfigParserFactoryImpl implements
```

```
              ConfigParserFactory {
   public final ConfigParser createConfigParser(
     final ConfigFormat configFormat
   ) {
     return switch(configFormat) {
       case JSON -> new JsonConfigParser();
       case YAML -> new YamlConfigParser();
       default ->
         throw new IllegalArgumentException(
           "Unsupported config format"
         );
    };
  }
}
```

You should follow the *program against interfaces principle* and use the abstract `ConfigParserFactory` in your code instead of a concrete factory. Then using the *dependency injection principle*, you can inject the wanted factory implementation, like `ConfigParserFactoryImpl`.

When unit testing code, you should create mock objects instead of real ones with a factory. The abstract factory pattern comes to your help because you can inject a mock instance of the `ConfigParserFactory` in the tested code. Then you can expect the mocked `createConfigParser` method to be called and return a mock instance of the `ConfigParser` interface. And then, you can expect the `parse` method to be called on the `ConfigParser` mock and return a mocked configuration. Below is an example unit test that uses JUnit5 and JMockit library. We test the `initialize` method in an `Application` class containing a `ConfigParserFactory` field. The `Application` class uses the `ConfigParserFactory` instance to create a `ConfigParser` to parse the application configuration. In the below test, we inject a `ConfigParserFactory` mock to an `Application` instance using the `@Injectable` annotation from JMockit. Unit testing and mocking are better described later in the *testing principles* chapter.

```
public class Application {
  private ConfigParserFactory configParserFactory;
  private Config config;

  public Application(final ConfigParserFactory configParserFactory) {
    this.configParserFactory = configParserFactory;
  }

  public void initialize() {
    // ...

    final var configParser = configParserFactory.createConfigParser(...);
    config = configParser.parse(...);

    // ...
  }

  public Config getConfig() {
    return config;
  }
```

```
}

public class ApplicationTests {
  @Tested
  Application application;

  @Injectable
  ConfigParserFactory configParserFactoryMock;

  @Mocked
  ConfigParser configParserMock;

  @Mocked
  Config configMock;

  @Test
  public void testInitialize() {
    // GIVEN
    new Expectations() {{
      configParserFactoryMock.createConfigParser(...);
      result = configParserMock;
      configParserMock.parse(...);
      result = configMock;
    }};

    // WHEN
    application.initialize();

    // THEN
    assertEquals(application.getConfig(), configMock);
  }
}
```

Factory Method Pattern

> *In the factory method pattern, objects are created using one or more static factory methods in a class, and the class constructor is made private.*

If you want to validate parameters in a constructor, the constructor may throw. You cannot return an error value from a constructor unless you use an *out* parameter, but that pattern is generally discouraged. Creating constructors that cannot throw is recommended because it is relatively easy to forget to catch errors thrown from a constructor if nothing in the constructor signature tells it can throw. Java is an exception to this rule when you throw a checked exception from a constructor. It is impossible to forget to handle checked exceptions. See the next chapter for a discussion about the *error/exception handling principle*.

Below is an example of a constructor that can throw:

```
class Url {
  constructor(
    scheme: string,
    port: number,
    host: string,
    path: string,
    query: string
  ) {
```

```
    // Validate the arguments and throw if invalid
  }
}
```

You can use the factory method pattern to overcome the problem of throwing an error from a constructor. You can make a factory method to return an optional value (if you don't need to return an error cause) or make the factory method throw. We can add a *try* prefix to the factory method name to signify that it can throw. Then, the function signature (function name) communicates to readers that the function can throw.

Below is an example class with two factory methods and a private constructor:

Url.ts

```
class Url {
  private constructor(
    scheme: string,
    port: number,
    host: string,
    path: string,
    query: string
  ) {
    // ...
  }

  static createUrl(
    scheme: string,
    port: number,
    host: string,
    path: string,
    query: string
 ): Url | null {
    // Validate the arguments and return 'null' if invalid
  }

  static tryCreateUrl(
    scheme: string,
    port: number,
    host: string,
    path: string,
    query: string
  ): Url {
    // Validate the arguments and throw if invalid
  }
}
```

Returning an optional value from a factory method allows utilizing functional programming techniques. Here is an example in Java:

```
public class Url {
  private Url(
    final String scheme,
    final String host,
    final int port,
    final String path,
    final String query
  ) {
    // ...
```

```
  }

  public static Optional<Url> createUrl(
    final String scheme,
    final String host,
    final int port,
    final String path,
    final String query
  ) {
    // ...
  }
}

final var maybeUrl = Url.createUrl(...);
maybeUrl.ifPresent(url -> {
  // Do something with the validated and correct 'url'
});
```

Builder Pattern

> *Builder pattern allows you to construct objects piece by piece.*

In the builder pattern, you add properties to the built object with `addXXX` methods of the builder class. After adding all the needed properties, you can build the final object using the `build` or `buildXXX` method of the builder class.

For example, you can construct a URL from the parts of the URL. Below is a Java example of using a `UrlBuilder` class:

```
final Optional<Url> url = new UrlBuilder()
  .addScheme("https")
  .addHost("www.google.com")
  .buildUrl();
```

The builder pattern has the benefit that properties given for the builder can be validated in the build method. You can make the builder's build method return an optional indicating whether the building was successful. Or, you can make the build method throw if you need to return an error. Then you should name the build method using a *try* prefix, for example, `tryBuildUrl`. The builder pattern also has the benefit of not needing to add default properties to the builder. For example, *https* could be the default scheme, and if you are building an HTTPS URL, the `addScheme` is not needed to be called. The only problem is that you must consult the builder documentation to determine the default values.

One drawback with the builder pattern is that you can give the parameters logically in the wrong order like this:

```
final Optional<Url> url = new UrlBuilder()
  .addHost("www.google.com")
  .addScheme("https")
  .buildUrl();
```

It works but does not look so nice. So if you are using a builder, always try to give the parameters for the builder in a logically correct order if such order exists. The builder pattern works well when there isn't any inherent order among the parameters. Below is an example of such a case: A house built with a `HouseBuilder` class.

```
final House house = new HouseBuilder()
  .addKitchen()
  .addLivingRoom()
  .addBedrooms(3)
  .addBathRooms(2)
  .addGarage()
  .buildHouse();
```

You can achieve functionality similar to a builder with a factory method with default parameters:

Url.ts

```
class Url {
  private constructor(
    host: string,
    path?: string,
    query?: string,
    scheme = 'https',
    port = 443
  ) {
    // ...
  }

  static createUrl(
    host: string,
    path?: string,
    query?: string,
    scheme = 'https',
    port = 443
  ): Url | null {
    // Validate the arguments and return 'null' if invalid
  }
}
```

In the factory method above, there is clear visibility of what the default values are. Of course, you cannot now give the parameters in a logical order. There is also a greater possibility that you accidentally provide some parameters in the wrong order because many of them are of the same type (string). This won't be a potential issue with a builder where you use a method with a specific name to give a specific parameter. In modern development environments, giving parameters in the wrong order is less probable because IDEs offer inlay parameter hints. It is easy to see if you provide a particular parameter in the wrong position. As shown below, giving parameters in the wrong order can also be avoided using semantically validated function parameter types. Semantically validated function parameters will be discussed later in this chapter.

Url.ts

```
class Url {
  static createUrl(
    host: Host,
    path?: Path,
```

```
    query?: Query,
    scheme = Scheme.createScheme('https'),
    port = Port.createPort(443)
 ): Url | null {
    // ...
  }
}
```

You can also use factory method overloading in languages like Java, where default parameters are not supported. But that solution, for example, in the `Url` class case, can not be easily implemented and requires quite many overloaded methods to be introduced, which can be overwhelming for a developer.

You can always use a parameter object, not only in Java but in many other languages, too. Below is an example in Java:

```
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class UrlParams {
  private String scheme = "https";
  private String host;
  private int port = 443;
  private String path = "";
  private String query = "";

  UrlParams(final String host) {
    this.host = host;
  }
}

public class Url {
  private Url(final UrlParams urlParams) {
    // ...
  }

  public static Optional<Url> createUrl(
    final UrlParams urlParams
  ) {
    // ...
  }
}

final var urlParams = new UrlParams("www.google.com");
urlParams.setQuery("query=design+patterns");
final var maybeUrl = Url.createUrl(urlParams);
```

The above solution is quite similar to using a builder, only slightly more verbose, of course.

Singleton Pattern

> *Singleton pattern defines that a class can have only one instance.*

Singletons are very common in pure object-oriented languages like Java. In many cases, a singleton class can be identified as not having any state. And this is why only one instance of the class is needed. There is no point in creating multiple instances that are the same. In some non-pure object-oriented languages, singletons are not as common as in pure object-oriented languages and can often be replaced by just defining functions.

In JavaScript/TypeScript, a singleton instance can be created in a module and exported. When you import the instance from the module in other modules, the other modules will always get the same exported instance, not a new instance every time. Below is an example of such a singleton:

myClassSingleton.ts

```
class MyClass {
  // ...
}

export const myClassSingleton = new MyClass();
```

otherModule.ts

```
import { myClassSingleton } from 'myClassSingleton';

// ...
```

The singleton pattern can be implemented using a static class because it cannot be instantiated. The problem with a static class is that the singleton class is then hardcoded, and static classes can be hard or impossible to mock in unit testing. We should remember to *program against interfaces*. The best way to implement the singleton pattern is by using the *dependency inversion principle* and the *dependency injection principle*. Below is an example in Java using the *Google Guice* library for handling dependency injection. The constructor of the `FileConfigReader` class expects a `ConfigParser`. We annotate the constructor with the `@Inject` annotation to inject an instance implementing the `ConfigParser` interface:

```
import com.google.inject.Inject;

public interface ConfigReader {
  Configuration tryRead(...);
}

public class FileConfigReader
         implements ConfigReader {
  private ConfigParser configParser;

  @Inject
  public FileConfigReader(
    final ConfigParser configParser
  ) {
    this.configParser = configParser;
  }

  public Configuration tryRead(
    final String configFilePathName
  ) {
    final String configFileContents = // Read configuration file
```

```
    final var configuration =
      configParser.tryParse(configFileContents);

    return configuration;
  }
}
```

In the below DI module, we configure a singleton with a lazy binding. In the lazy binding, the `JsonConfigParser` class is only created when needed to be used.

DiModule.java

```
import com.google.inject.AbstractModule;

public class DiModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(ConfigParser.class)
      .to(JsonConfigParser.class)
      .in(Scopes.SINGLETON);
  }
}
```

Alternatively, we can define an eager singleton:

DiModule.java

```
import com.google.inject.AbstractModule;

public class DiModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(ConfigParser.class)
      .to(JsonConfigParser.class)
      .asEagerSingleton();
  }
}
```

Prototype Pattern

> *The prototype pattern lets you create a new object using an existing object as a prototype.*

Let's have an example with a `DrawnShape` class:

```
public interface Shape {
  // ...
}

// Implement concrete shapes...

public interface Position {
  int getX();
  int getY();
}

public class DrawnShape {
```

```
  private final Position position;
  private final Shape shape;

  public DrawnShape(
    final Position position,
    final Shape shape
  ) {
    this.position = position;
    this.shape = shape;
  }

  public DrawnShape(
    final Position position,
    final DrawnShape drawnShape
  ) {
    this.position = position;
    shape = drawnShape.getShape();
  }

  public DrawnShape cloneTo(
    final Position position
  ) {
    return new DrawnShape(position, this);
  }

  public Shape getShape() {
    return this.shape;
  }
}
```

In the second constructor, we are using the prototype pattern. A new `DrawnShape` object is created from an existing `DrawnShape` object. An alternative way to use the prototype pattern is to call the `cloneTo` method on a prototype object and give the position parameter to specify where the new shape should be positioned.

The prototype pattern is also used in JavaScript to implement prototypal inheritance. Since ECMAScript version 6, class-based inheritance has been available, and prototypal inheritance is not needed to be used.

The idea of prototypal inheritance is that the common parts for the same class objects are stored in a prototype instance. These common parts typically mean the shared methods. There is no sense in storing the methods multiple times in each object. That would be a waste of resources because Javascript functions are objects themselves.

When you create a new object with the `Object.create` method, you give the prototype as a parameter. After that, you can set properties for the newly created object. When you call a method on the created object, and if that method is not found in the object's properties, the prototype object will be looked up for the method. Prototypes can be chained so that a prototype object contains another prototype object. This chaining is used to implement an inheritance chain.

Below is a simple example of prototypal inheritance:

```
const pet = {
  name: '',
  getName: function() { return this.name; }
};

// Creates a new object with 'pet' object as a prototype
const petNamedBella = Object.create(pet);

petNamedBella.name = 'Bella';
console.log(petNamedBella.getName()); // Prints 'Bella'

// Prototype of a dog which contains 'pet' as nested prototype
const dog = {
  bark: function() { console.log('bark'); },
  __proto__: pet
}

// Creates a new object with 'dog' object as prototype
const dogNamedLuna = Object.create(dog);

dogNamedLuna.name = 'Luna';
console.log(dogNamedLuna.getName()); // Prints 'Luna'
dogNamedLuna.bark(); // Prints 'bark'
```

Object Pool Pattern

> *In the object pool pattern, created objects are stored in a pool where objects can be acquired from and returned for reuse. The object pool pattern is an optimization pattern because it allows the reuse of created objects.*

If you need to create many short-lived objects, you should utilize an object pool and reduce the need for memory allocation and de-allocation, which takes time. In garbage-collected languages, frequent object creation and deletion cause extra work for the garbage collector, which consumes CPU time.

Below is an example object pool implementation in C++. The below `LimitedSizeObjectPool` class implementation uses a spin lock in its methods to achieve thread safety. More about thread safety in the coming *concurrent programming principles* chapter.

ObjectPool.h

```
#include <memory>

template <typename T>
class ObjectPool
{
public:
  virtual ~ObjectPool() = default;

  virtual std::shared_ptr<T> acquireObject() = 0;
  virtual void returnObject(std::shared_ptr<T> object) = 0;
};
```

LimitedSizeObjectPool.h

```
#include <deque>
#include "ScopedSpinlock.h"
#include "Spinlock.h"
#include "ObjectPool.h"

template <typename T>
class LimitedSizeObjectPool : public ObjectPool<T>
{
public:
  explicit LimitedSizeObjectPool(const size_t maxPoolSize):
    m_maxPoolSize(maxPoolSize)
  {}

  std::shared_ptr<T> acquireObject()
  {
    std::shared_ptr<T> object;
    const ScopedSpinlock scopedLock{m_lock};

    if (m_pooledObjects.empty())
    {
      object = std::make_shared<T>();
    }
    else
    {
      object = m_pooledObjects.front();
      m_pooledObjects.pop_front();
    }

    return object;
  }

  void returnObject(std::shared_ptr<T> object)
  {
    const ScopedSpinlock scopedLock{m_lock};

    const bool poolIsFull =
      m_pooledObjects.size() >= m_maxPoolSize;

    if (poolIsFull)
    {
      object.reset();
    }
    else
    {
      m_pooledObjects.push_back(object);
    }
  }

private:
  Spinlock m_lock;
  size_t m_maxPoolSize;
  std::deque<std::shared_ptr<T>> m_pooledObjects;
};
```

Below is a slightly different implementation of an object pool. The below implementation accepts clearable objects, meaning objects returned to the pool are cleared before reusing. The below implementation allows you to define whether the allocated objects are wrapped inside a shared or unique pointer. You can also supply parameters used when constructing an object.

ObjectPool.h

```
#include <concepts>
#include <deque>
#include <memory>

template<typename T>
concept ClearableObject =
requires(T object)
{
  { object.clear() } -> std::convertible_to<void>;
};

template<typename T, typename U>
concept Pointer = std::derived_from<T, std::shared_ptr<U>> ||
                  std::derived_from<T, std::unique_ptr<U>>;

template<
  ClearableObject O,
  typename ObjectInterface,
  Pointer<ObjectInterface> OP,
  typename ...Args
>
class ObjectPool
{
public:
  virtual ~ObjectPool() = default;

  virtual OP acquireObject(Args&& ...args) = 0;

  virtual void acquireObjects(
    std::deque<OP>& objects,
    size_t objectCount,
    Args&& ...args
   ) = 0;

  virtual void returnObject(OP object) = 0;
  virtual void returnObjects(std::deque<OP>& objects) = 0;
};
```

LimitedSizeObjectPool.h

```
#include "ScopedLock.h"
#include "Spinlock.h"
#include "ObjectPool.h"

template<
  ClearableObject O,
  typename ObjectInterface,
  Pointer<ObjectInterface> OP,
  typename ...Args
>
class LimitedSizeObjectPool :
   public ObjectPool<O, ObjectInterface, OP, Args...>
{
public:
  explicit LimitedSizeObjectPool(const size_t maxPoolSize) :
    m_maxPoolSize(maxPoolSize)
  {}

  OP acquireObject(Args&& ...args) override
  {
    const ScopedLock scopedLock(m_lock);
```

```
    OP acquiredObject;

    if (const bool poolIsEmpty = m_pooledObjects.empty();
            poolIsEmpty)
    {
      acquiredObject = OP{new O{std::forward<Args>(args)...}};
    }
    else
    {
      acquiredObject = m_pooledObjects.front();
      m_pooledObjects.pop_front();
    }

    return acquiredObject;
  }

  void acquireObjects(
    std::deque<OP>& objects,
    const size_t objectCount,
    Args&& ...args
  ) override
  {
    for (size_t n{1U}; n <= objectCount; ++n)
    {
      objects.push_back(acquireObject(std::forward<Args>(args)...));
    }
  }

  void returnObject(OP object) override
  {
    const ScopedLock scopedLock(m_lock);

    if (const bool poolIsFull = m_pooledObjects.size() >=
                                m_maxPoolSize;
            poolIsFull)
    {
        object.reset();
    }
    else
    {
        object->clear();
        m_pooledObjects.push_back(object);
    }
  }

  void returnObjects(std::deque<OP>& objects) override
  {
    while (!objects.empty())
    {
      returnObject(objects.front());
      objects.pop_front();
    }
  }

private:
  size_t m_maxPoolSize;
  Spinlock m_lock;
  std::deque<OP> m_pooled
```

In the below example, we create a message pool for a maximum of 5000 output messages. We get a shared pointer to an output message from the pool. The pool's concrete class to create new objects

is `OutputMessageImpl`. When we acquire an output message from the pool, we provide a `size_t` type value (= output message length) to the constructor of the `OutputMessageImpl` class. The `OutputMessageImpl` class must be clearable, i.e., it must have a `clear` method returning `void`.

```
LimitedSizeObjectPool<
  OutputMessageImpl,
  OutputMessage,
  std::shared_ptr<OutputMessage>,
  size_t
> outputMessagePool{5000U};

// Acquire an output message of 1024 bytes from the pool.
const auto outputMessage = outputMessagePool.acquireObject(1024U);
```

Structural Design Patterns

This section describes structural design patterns. Most patterns use object composition as the primary method to achieve a particular design. The following design patterns are presented:

- Composite pattern
- Facade pattern
- Bridge pattern
- Strategy pattern
- Adapter pattern
- Proxy pattern
- Decorator pattern
- Flyweight pattern

Composite Pattern

> *In the composite pattern, a class can be composed of itself, i.e., the composition is recursive.*

Recursive object composition can be depicted by how a user interface can be composed of different widgets. In the example below, we have a `Pane` class that is a `Widget`. A `Pane` object can contain several other `Widget` objects, meaning a `Pane` object can contain other `Pane` objects.

```
interface Widget {
  void render();
}

public class Pane implements Widget {
  private final List<Widget> widgets;

  public void render() {
    // Render each widget inside pane
  }
}
```

```
public class StaticText implements Widget {
  public void render() {
   // Render static text widget
  }

  // ...
}

public class TextInput implements Widget {
  public void render() {
    // Render text input widget
  }
}

public class Button implements Widget {
  public void render() {
    // Render button widget
  }
}

public class UIWindow {
  private final List<Widget> widgets = new ArrayList<>(10);

  public void render() {
    widgets.forEach(Widget::render);
  }
}
```

Objects that form a tree structure are composed of themselves recursively. Below is an Avro record field schema with a nested record field:

```
{
  "type": "record",
  "name": "sampleMessage",
  "fields": [
    {
      "name": "field1",
      "type": "string"
    },
    {
      "name": "nestedRecordField",
      "namespace": "nestedRecordField",
      "type": "record",
      "fields": [
        {
          "name": "nestedField1",
          "type": "int",
          "signed": "false"
        }
      ]
    }
  ]
}
```

For parsing an Avro schema, we could define classes for different sub-schemas by the field type. When analyzing the below example, we can notice that the `RecordAvroFieldSchema` class can contain any `AvroFieldSchema` object, also other `RecordAvroFieldSchema` objects, making a

`RecordAvroFieldSchema` object a composite object.

```
public interface AvroFieldSchema {
  // ...
}

public class RecordAvroFieldSchema implements AvroFieldSchema {
  private final List<AvroFieldSchema> subFieldSchemas;

  // ...
}

public class StringAvroFieldSchema implements AvroFieldSchema {
  // ...
}

public class IntAvroFieldSchema implements AvroFieldSchema {
  // ...
}
```

Facade Pattern

> *In the facade pattern, an object on a higher level of abstraction is composed of objects on a lower level of abstraction. The higher-level object acts as a facade in front of the lower-level objects. Lower-level objects behind the facade are either only or mainly only accessible by the facade.*

Let's use the data exporter microservice as an example. For that microservice, we could create a `Configuration` interface that can be used to obtain configuration for the different parts (input, transformer, and output) of the data exporter microservice. The `Configuration` interface acts as a facade. Users of the facade need not see behind the facade. They don't know what happens behind the facade. And they shouldn't care because they are just using the interface provided by the facade.

There can be various classes doing the actual work behind the facade. In the below example, there is a `ConfigReader` that reads configuration from possibly different sources (from a local file or a remote service, for example) and there are configuration parsers that can parse a specific part of the configuration, possibly in different data formats like JSON or YAML. None of these implementations and details are visible to the user of the facade. Any of these implementations behind the facade can change at any time without affecting the users of the facade because facade users are not coupled to the lower-level implementations.

Below is the implementation of the `Configuration` facade in Java:

```
import com.google.inject.Inject;

public interface Configuration {
  InputConfig tryGetInputConfig();
  TransformerConfig tryGetTransformerConfig();
  OutputConfig tryGetOutputConfig();
}
```

```
public class ConfigurationImpl implements Configuration {
  private final ConfigReader configReader;
  private final InputConfigParser inputConfigParser;
  private final TransformerConfigParser transformerConfigParser;
  private final OutputConfigParser outputConfigParser;
  private String configString = "";
  private Optional<InputConfig> inputConfig = Optional.empty();
  private Optional<OutputConfig> outputConfig = Optional.empty();

  private Optional<TransformerConfig> transformerConfig =
    Optional.empty();

  @Inject
  public ConfigurationImpl(
    final ConfigReader configReader,
    final InputConfigParser inputConfigParser,
    final TransformerConfigParser transformerConfigParser,
    final OutputConfigParser outputConfigParser
  ) {
    // ...
  }

  public InputConfig tryGetInputConfig() {
    return inputConfig.orElseGet(() -> {
      tryReadConfigIfNeeded();

      inputConfig =
        inputConfigParser.tryParseInputConfig(configString);

      return inputConfig;
    });
  }

  public TransformerConfig tryGetTransformerConfig() {
    // ...
  }

  public OutputConfig tryGetOutputConfig() {
    // ...
  }

  private void tryReadConfigIfNeeded() {
    if (configString.isEmpty()) {
      configString =
        configReader.tryRead(...);
    }
  }
}
```

There is a unique alternative available when implementing the above facade in Java: only the `Configuration` interface and the `ConfigurationImpl` class could be made public, and all the configuration reading and parsing related interfaces and classes could be package-private. This would make the usage of the facade mandatory. No one else except the `ConfigurationImpl` class could use the lower-level implementation classes related to configuration reading and parsing.

Bridge Pattern

> *In the bridge pattern, the implementation of a class is delegated to another class. The original class is "abstract" in the sense that it does not have any behavior except the delegation to another class, or it can have some higher level control logic on how it delegates to another class.*

Don't confuse the word "abstract" here with an abstract class. In an abstract class, some behavior is not implemented at all, but the implementation is deferred to subclasses of the abstract class. Here, instead of the term "abstraction class", we could use the term *delegating class* instead.

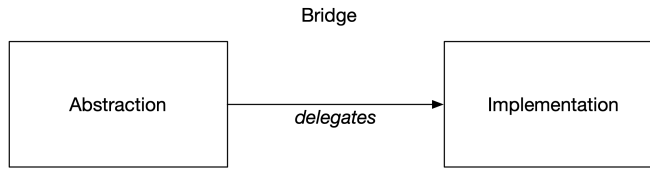

Fig 3.5 Bridge Pattern

Let's have an example with shapes and drawings capable of drawing different shapes:

```
public interface Shape {
  void render(final ShapeRenderer renderer);
}

public class RectangleShape implements Shape {
  private final Point upperLeftCorner;
  private final int width;
  private final int height;

  public RectangleShape(
    final Point upperLeftCorner,
    final int width,
    final int height
    ) {
    this.upperLeftCorner = upperLeftCorner;
    this.width = width;
    this.height = height;
  }

  public void render(final ShapeRenderer renderer) {
    renderer.renderRectangleShape(upperLeftCorner, width, height);
  }
}

public class CircleShape implements Shape {
  private final Point center;
  private final int radius;

  public CircleShape(final Point center, final int radius) {
    this.center = center;
    this.radius = radius;
  }
```

```
  public void render(final ShapeRenderer renderer) {
    renderer.renderCircleShape(center, radius);
  }
}
```

The above `RectangleShape` and `CircleShape` classes are abstractions because they delegate their functionality (rendering) to an external class (implementation class) of the `ShapeRenderer` type. We can provide different rendering implementations for the shape classes. Let's define two shape renderers, one for rendering raster shapes and another for rendering vector shapes:

```
public interface ShapeRenderer {
  void renderCircleShape(final Point center, final int radius);

  void renderRectangleShape(
    final Point upperLeftCorner,
    final int width,
    final int height
  );

  // Methods for rendering other shapes...
}

public class RasterShapeRenderer implements ShapeRenderer {
  private final Canvas canvas;

  public RasterShapeRenderer(final Canvas canvas) {
    this.canvas = canvas;
  }

  public void renderCircleShape(
    final Point center,
    final int radius
  ) {
    // Renders circle to canvas
  }

  public void renderRectangleShape(
    final Point upperLeftCorner,
    final int width,
    final int height
  ) {
    // Renders a rectangle to canvas
  }

  // Methods for rendering other shapes to the canvas
}

public class VectorShapeRenderer implements ShapeRenderer {
  private final SvgElement svgRoot;

  public VectorShapeRenderer(final SvgElement svgRoot) {
    this.svgRoot = svgRoot;
  }

  public void renderCircleShape(
    final Point center,
    final int radius
  ) {
    // Render circle as SVG element and attach as child to SVG root
```

```
  }

  public void renderRectangleShape(
    final Point upperLeftCorner,
    final int width,
    final int height
  ) {
    // Render rectangle as SVG element
    // and attach as child to SVG root
  }

  // Methods for rendering other shapes
}
```

Let's implement two different drawings, a raster, and a vector drawing:

```
public interface Drawing {
  ShapeRenderer getShapeRenderer();
  void draw();
  void save();
}

public abstract class AbstractDrawing implements Drawing {
  private final String name;

  public AbstractDrawing(final String name) {
    this.name = name;
  }

  public abstract ShapeRenderer getShapeRenderer();
  public abstract String getFileExtension();
  public abstract byte[] getData();

  public void save() {
    final var fileName = name + getFileExtension();
    final var data = getData();

    // Save the 'data' to 'fileName'
  }

  public void draw(final List<Shape> shapes) {
    for (final var shape: shapes) {
      shape.render(getShapeRenderer());
    }
  }
}

public class RasterDrawing extends AbstractDrawing {
  private final Canvas canvas = new Canvas();

  private final RasterShapeRenderer shapeRenderer =
    new RasterShapeRenderer(canvas);

  public RasterDrawing(final String name) {
    super(name);
  }

  public ShapeRenderer getShapeRenderer() {
    return shapeRenderer;
  }

  public String getFileExtension() {
```

```
    return ".png";
  }

  public byte[] getData() {
    // get data from the 'canvas' object
  }
}

public class VectorDrawing extends AbstractDrawing {
  private final SvgElement svgRoot = new SvgElement();

  private final VectorShapeRenderer shapeRenderer =
    new VectorShapeRenderer(svgRoot);

  public VectorDrawing(final String name) {
    super(name);
  }

  public ShapeRenderer getShapeRenderer() {
    return shapeRenderer;
  }

  public String getFileExtension() {
    return ".svg";
  }

  public byte[] getData() {
    // get data from the 'svgRoot' object
  }
}
```

In the above example, we have delegated the rendering behavior of the shape classes to concrete classes implementing the `ShapeRenderer` interface. The `Shape` classes only represent a shape but don't render the shape. They have a single responsibility of representing a shape. Regarding rendering, the shape classes are "abstractions" because they delegate the rendering to other classes responsible for rendering different shapes.

Now we can have a list of shapes and render them differently. We can do this as shown below because we did not couple the shape classes with any specific rendering behavior.

```
final List<Shape> shapes = new ArrayList<>(50);
// Add various shapes to 'shapes' list here...

final var rasterDrawing = new RasterDrawing("raster-drawing-1");
rasterDrawing.draw(shapes);
rasterDrawing.save();

final var vectorDrawing = new VectorDrawing("vector-drawing-1");
vectorDrawing.draw(shapes);
vectorDrawing.save();
```

Strategy Pattern

> *In the strategy pattern, the functionality of an object can be changed by changing an instance of a composed type to a different instance of that type.*

Below is an example where the behavior of a `ConfigReader` class can be changed by changing the value of the `configParser` field to an instance of a different class. The default behavior is to parse the configuration in JSON format, which can be achieved using the default constructor.

```
public class ConfigReader {
  private final ConfigParser configParser;

  public ConfigReader() {
    configParser = new JsonConfigParser();
  }

  public ConfigReader(final ConfigParser configParser) {
    this.configParser = configParser;
  }

  public Configuration tryRead(final String configFilePathName) {
    // Try read the configuration file contents to a string
    // variable named 'configFileContents'

    final Configuration configuration =
      configParser.tryParse(configFileContents);

    return configuration;
  }
}
```

Using the strategy pattern, we can change the functionality of a `ConfigReader` class object by changing the `configParser` field value. For example, there could be the following classes available that implement the `ConfigParser` interface:

- `JsonConfigParser`
- `YamlConfigParser`
- `TomlConfigParser`

We can dynamically change the behavior of the `ConfigReader` class to use a YAML parsing strategy by giving an instance of the `YamlConfigParser` class as a parameter for the `ConfigReader` constructor.

Adapter Pattern

> *The adapter pattern changes one interface to another interface. The adapter pattern allows you to adapt different interfaces to a single interface.*

In the below example, we have defined a `Message` interface for messages that can be consumed from a data source using a `MessageConsumer`.

Message.h

```
#include <cstdint>

class Message
{
public:
  Message() = default;
  virtual ~Message() = default;

  virtual uint8_t* getData() const = 0;
  virtual std::size_t getDataLengthInBytes() const = 0;
};
```

MessageConsumer.h

```
#include <memory>
#include "Message.h"

class MessageConsumer
{
public:
  MessageConsumer() = default;
  virtual ~MessageConsumer() = default;

  virtual std::shared_ptr<Message> consumeMessage() = 0;
};
```

Next, we can define the message and message consumer adapter classes for Apache Kafka and Apache Pulsar:

KafkaMessageConsumer.h

```
#include "MessageConsumer.h"

class KafkaMessageConsumer : public MessageConsumer
{
public:
  KafkaMessageConsumer(...);
  ~KafkaMessageConsumer() override;

  std::shared_ptr<Message> consumeMessage() override {
    // Consume a message from Kafka using a 3rd party
    // Kafka library, e.g. LibRdKafka
    // Wrap the consumed LibRdKafka message inside an instance
    // of KafkaMessage class
    // Return the KafkaMessage instance
  }
};
```

KafkaMessage.h

```
#include <bit>
#include <librdkafka/rdkafkacpp.h>
#include "Message.h"
```

```
class KafkaMessage : public Message
{
public:
  explicit KafkaMessage(RdKafka::Message* const message):
    m_message(message)
  {}

  ~KafkaMessage() override
  {
    delete m_message;
  }

  uint8_t* getData() const override
  {
    return std::bit_cast<uint8_t*>(m_message->payload());
  }

  std::size_t getDataLengthInBytes() const override
  {
    return m_message->len();
  }

private:
  RdKafka::Message* m_message;
};
```

PulsarMessageConsumer.h

```
#include "MessageConsumer.h"

class PulsarMessageConsumer : public MessageConsumer
{
public:
  PulsarMessageConsumer(...);
  ~PulsarMessageConsumer() override;

  std::shared_ptr<Message> consumeMessage() override {
    // Consume a message from Pulsar using the Pulsar C++ client
    // Wrap the consumed Pulsar message inside an instance
    // of PulsarMessage
    // Return the PulsarMessage instance
  }
};
```

PulsarMessage.h

```
#include "Message.h"

class PulsarMessage : public Message
{
public:
  // ...
};
```

Now we can use Kafka or Pulsar data sources with identical consumer and message interfaces. In the future, it will be easy to integrate a new data source into the system. We only need to implement appropriate adapter classes (message and consumer classes) for the new data source. No other code changes are required. Thus, we would be following the *open-closed principle* correctly.

Let's imagine that the API of the used *LibRdKafka* library changed. We don't need to make changes in many places in the code. We need to create new adapter classes (message and consumer classes) for the new *LibRdKafka* API and use those new adapter classes in place of the old adapter classes. All this work is again following the *open-closed principle*.

Consider using the adapter pattern even if there is nothing to adapt to, especially when working with 3rd party libraries. Because then you will be prepared for the future when changes can come. It might be possible that a 3rd party library interface changes or there is a need to take a different library into use. If you have not used the adapter pattern, taking a new library or library version into use could mean that you must make many small changes in several places in the codebase, which is error-prone and against the *open-closed principle*.

Let's have an example of using a 3rd party logging library. Initially, our adapter for the *abc-logging-library* is just a wrapper around the `abcLogger` instance from the library. There is not any actual adapting done.

logger.ts

```
import abcLogger from 'abc-logging-library';
import { LogLevel } from 'LogLevel';

interface Logger {
  log(logLevel: LogLevel, logMessage: string): void;
}

class AbcLogger implements Logger {
  log(logLevel: LogLevel, logMessage: string): void {
    abcLogger.log(logLevel, logMessage);
  }
}

export default new AbcLogger();
```

Suppose that in the future, a better logging library is available called *xyz-logging-library*, and we would like to take that into use, but it has a bit different interface. Its logging instance is called `xyzLogWriter`, the logging method is named differently, and the parameters are given in different order compared to the *abc-logging-library*. We can create an adapter for the new logging library, and no other code changes are required elsewhere in the codebase:

logger.ts

```
import xyzLogWriter from 'xyz-logging-library';
import { LogLevel } from 'LogLevel';

interface Logger {
  log(logLevel: LogLevel, logMessage: string): void;
}

class XyzLogger implements Logger {
  log(logLevel: LogLevel, logMessage: string): void {
    xyzLogWriter.writeLogEntry(logMessage, logLevel);
  }
}
```

```
export default new XyzLogger();
```

We don't have to modify all the places in the code where logging is used. And usually, logging is used in many places. We have saved ourselves from a lot of error-prone and unnecessary work, and once again, we have followed the *open-closed principle*.

Proxy Pattern

The proxy pattern enables conditionally modifying or augmenting the behavior of an object.

When using the proxy pattern, you define a proxy class that wraps another class (the proxied class). The proxy class conditionally delegates to the wrapped class. The proxy class implements the interface of the wrapped class and is used in place of the wrapped class in the code.

Below is an example of a TypeScript proxy class, `CachingEntityStore`, that caches the results of entity store operations:

```
class MemoryCache<K, V> {
  // ...

  retrieveBy(key: K): V {
    // ...
  }

  store(key: K, value: V, timeToLiveInSecs?: number): void {
    // ...
  }
}

interface EntityStore<T> {
  getEntityById(id: number): Promise<T>;
}

class DbEntityStore<T> implements EntityStore<T> {
  getEntityById(id: number): Promise<T> {
    // Try get entity from database
  }
}

class CachingEntityStore<T> implements EntityStore<T> {
  private readonly entityCache = new MemoryCache<number, T>();

  constructor(private readonly entityStore: EntityStore<T>)
  {}

  async getEntityById(id: number): Promise<T> {
    let entity = this.entityCache.retrieveBy(id);

    if (entity === undefined) {
      entity = await this.entityStore.getEntityById(id);
      const timeToLiveInSecs = 60;
      this.entityCache.store(id, entity, timeToLiveInSecs);
    }

    return entity;
```

```
  }
}
```

In the above example, the `CachingEntityStore` class is the proxy class wrapping an `EntityStore`. The proxy class is modifying the wrapped class behavior by conditionally delegating to the wrapped class. It delegates to the wrapped class only if an entity is not found in the cache.

Below is another TypeScript example of a proxy class that authorizes a user before performing a service operation:

```
interface UserService {
  getUserById(id: number): Promise<User>;
}

class UserServiceImpl implements UserService {
  getUserById(id: number): Promise<User> {
    // Try get user by id
  }
}

class AuthorizingUserService implements UserService {
  constructor(
    private readonly userService: UserService,
    private readonly userAuthorizer: UserAuthorizer
  ) {}

  async getUserById(id: number): Promise<User> {
    try {
      await this.userAuthorizer.tryAuthorizeUser(id);
    } catch (error: any) {
      throw new UserServiceError(error.message);
    }

    return this.userService.getUserById(id);
  }
}
```

In the above example, the `AuthorizingUserService` class is a proxy class that wraps a `UserService`. The proxy class is modifying the wrapped class behavior by conditionally delegating to the wrapped class. It will delegate to the wrapped class only if authorization is successful.

Decorator Pattern

> *The decorator pattern enables augmenting the functionality of a class method(s) without the need to modify the class method(s).*

A decorator class wraps another class whose functionality will be augmented. The decorator class implements the interface of the wrapped class and is used in place of the wrapped class in the code. The decorator pattern is useful when you cannot modify an existing class, e.g., the existing class is in a 3rd party library. The decorator pattern also helps to follow the *open-closed principle* because you don't have to modify an existing method to augment its functionality. You can create a decorator class that contains the new functionality.

Below is a TypeScript example of the decorator pattern. There is a standard SQL statement executor implementation and two decorated SQL statement executor implementations: one that adds logging functionality and one that adds SQL statement execution timing functionality. Finally, a double-decorated SQL statement executor is created that logs an SQL statement and times its execution.

```
import logger from 'logger';
import { LogLevel } from 'LogLevel';

interface SqlStatementExecutor {
  tryExecute(
    sqlStatement: string,
    parameterValues?: any[]
  ): Promise<any>;
}

class SqlStatementExecutorImpl implements SqlStatementExecutor {
  // Implement getConnection()

  tryExecute(
    sqlStatement: string,
    parameterValues?: any[]
  ): Promise<any> {
    return this.getConnection().execute(sqlStatement,
                                        parameterValues);
  }
}

class LoggingSqlStatementExecutor
        implements SqlStatementExecutor {
  constructor(
    private readonly sqlStatementExecutor: SqlStatementExecutor
  ) {}

  tryExecute(
    sqlStatement: string,
    parameterValues?: any[]
  ): Promise<any> {
    logger.log(LogLevel.Debug,
                `Executing SQL statement: ${sqlStatement}`);

    return this.sqlStatementExecutor
      .tryExecute(sqlStatement, parameterValues);
  }
}

class TimingSqlStatementExecutor
        implements SqlStatementExecutor {
  constructor(
    private readonly sqlStatementExecutor: SqlStatementExecutor
  ) {}

  async tryExecute(
    sqlStatement: string,
    parameterValues?: any[]
  ): Promise<any> {
    const startTimeInMs = Date.now();

    const result =
      await this.sqlStatementExecutor
```

```
        .tryExecute(sqlStatement, parameterValues);

    const endTimeInMs = Date.now();
    const durationInMs = endTimeInMs - startTimeInMs;

    logger.log(LogLevel.Debug,
    `SQL statement execution duration: ${durationInMs} ms`);

    return result;
  }
}

const timingAndLoggingSqlStatementExecutor =
  new LoggingSqlStatementExecutor(
    new TimedSqlStatementExecutor(
      new SqlStatementExecutorImpl()));
```

Flyweight Pattern

> *The flyweight pattern is a memory-saving optimization pattern where flyweight objects reuse objects.*

Let's have a simple example with a game where different shapes are drawn at different positions. Let's assume that the game draws a lot of similar shapes but in different positions so that we can notice the difference in memory consumption after applying this pattern.

Shapes that the game draws have the following properties: size, form, fill color, stroke color, stroke width, and stroke style.

```
public interface Shape {
  // ...
}

// Color...
// StrokeStyle...

public class AbstractShape implements Shape {
  private final Color fillColor;
  private final Color strokeColor;
  private final int strokeWidth;
  private final StrokeStyle strokeStyle;

  // ...
}

public class CircleShape extends AbstractShape {
  private final int radius;

  // ...
}

// LineSegment...

public class PolygonShape extends AbstractShape {
  private final List<LineSegment> lineSegments;

  // ...
}
```

When analyzing the `PolygonShape` class, we can notice that it contains many properties that consume memory. Especially a polygon that has many line segments can consume a noticeable amount of memory. If the game draws many identical polygons in different screen positions and always creates a new `PolygonShape` object, there would be a lot of identical `PolygonShape` objects in the memory. To remediate this, we can introduce a flyweight class, `DrawnShapeImpl`, which contains the position of a shape and a reference to the actual shape. In this way, we can draw a lot of `DrawnShapeImpl` objects that all contain a reference to the same `PolygonShape` object:

```
public interface DrawnShape {
  // ...
}

public class DrawnShapeImpl implements DrawnShape {
  private final Shape shape;
  private Position screenPosition;

  public DrawnShapeImpl(
    final Shape shape,
    final Position screenPosition
  ) {
    this.shape = shape;
    this.screenPosition = screenPosition;
  }

  // ...
}

final Shape polygon = new PolygonShape(...);
final List<Position> positions = generateLotsOfPositions();

final var drawnPolygons = positions.stream().map(position ->
  new DrawnShapeImpl(polygon, position)
);
```

Behavioral Design Patterns

Behavioral design patterns describe ways to implement new behavior using object-oriented design. The following behavioral design patterns will be presented in the following sections:

- Chain of responsibility pattern
- Observer pattern
- Command/Action pattern
- Iterator pattern
- State pattern
- Mediator pattern
- Template method pattern
- Memento pattern
- Visitor pattern

- Null object pattern

Chain of Responsibility Pattern

The chain of responsibility pattern lets you pass requests along a chain of handlers.

When receiving a request, each handler can decide what to do:

- Process the request and then pass it to the next handler in the chain
- Process the request without passing it to the subsequent handlers (terminating the chain)
- Leave the request unprocessed and pass it to the next handler

One of the most famous implementations of this pattern is Java servlet filters. A servlet filter processes incoming HTTP requests before passing them to the actual servlet for handling. Servlet filters can be used to implement various functionality like logging, compression, encryption/decryption, input validation, etc.

Let's have an example of a servlet filter that adds logging before and after each HTTP request is processed:

LoggingFilter.java

```
import java.io.IOException;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;

@WebFilter(urlPatterns = {"/*"})
public class LoggingFilter implements Filter {
  // No initialization needed, thus empty method
  public void init(final FilterConfig filterConfig)
    throws ServletException
  {}

  // No cleanup needed, thus empty method
  public void destroy()
  {}

  public void doFilter(
    final ServletRequest request,
    final ServletResponse response,
    final FilterChain filterChain
  ) throws IOException, ServletException
  {
    final var responseWriter = response.getWriter();
    responseWriter.print("Before response\n");

    // Sends request to the next filter
    // or when no more filters to the servlet
    filterChain.doFilter(request, response);

    responseWriter.print("\nAfter response");
  }
}
```

HelloWorldServlet.java

```
import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/helloworld")
public class HelloWorldServlet extends HttpServlet {
  public void doGet(
    HttpServletRequest request,
    HttpServletResponse response
  ) throws ServletException, IOException {
    response.setContentType("text/plain");
    final var responseWriter = response.getWriter();
    responseWriter.print("Hello, world!");
  }
}
```

When we send an HTTP GET request to the */helloworld* endpoint, we should get the following response:

```
Before response
Hello, world!
After response
```

Let's implement a JWT authorization filter:

JwtAuthorizationFilter.java

```
import java.io.IOException;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletResponse;

@WebFilter(urlPatterns = {"/*"})
public class AuthorizationFilter implements Filter {
  public void init(final FilterConfig filterConfig)
    throws ServletException
  {}

  public void destroy()
  {}

  public void doFilter(
    final ServletRequest request,
    final ServletResponse response,
    final FilterChain filterChain
  ) throws IOException, ServletException {

    // From request's 'Authorization' header,
    // extract the bearer JWT
    // Set 'tokenIsPresent' variable value
    // to true or false
    // Verify the validity of JWT and assign result
    // to 'tokenIsValid' variable

    HttpServletResponse httpResponse =
```

```
      (HttpServletResponse) response;

    if (tokenIsValid) {
      filterChain.doFilter(request, response);
    } else if (tokenIsPresent) {
      // NOTE! filterChain is not invoked,
      // this will terminate the request
      httpResponse.setStatus(403);
      final var responseWriter = response.getWriter();
      responseWriter.print("Unauthorized");
      responseWriter.close();
    } else {
    // NOTE! filterChain is not invoked,
    // this will terminate the request
    httpResponse.setStatus(401);
    final var responseWriter = response.getWriter();
    responseWriter.print("Unauthenticated");
    responseWriter.close();
    }
  }
}
```

The Express framework for Node.js utilizes the chain of responsibility pattern for handling requests. In the Express framework, you can write pluggable behavior using *middlewares*, a concept similar to servlet filters in Java. Below is the same logging and authorization example as above, but written using JavaScript and the Express framework:

```
const express = require('express')
const app = express()

// Authorization middleware
function authorize(request, response, next) {
  // From request's 'Authorization' header,
  // extract the bearer JWT, if present
  // Set 'tokenIsPresent' variable value
  // Verify the validity of JWT and assign result
  // to 'tokenIsValid' variable

  if (tokenIsValid) {
    next();
  } else if (tokenIsPresent) {
    // NOTE! next is not invoked,
    // this will terminate the request
    response.writeHead(403);
    response.end('Unauthorized');
  } else {
    // NOTE! next is not invoked,
    // this will terminate the request
    response.writeHead(401);
    response.end('Unauthenticated');
  }
}

// Logging before middleware
function logBefore(request, response, next) {
  response.write('Before response\n');
  next();
}

// Use authorization and logging middlewares
```

```
app.use(authorize, logBefore);

app.get('/helloworld', (request, response, next) => {
  response.write('Hello World!\n');
  next();
});

// Logging after middleware
function logAfter(request, response, next) {
  response.write('After response\n');
  response.end();
  next();
}

app.use(logAfter);

app.listen(4000);
```

We cannot use Express middlewares as described below:

```
// Logging middleware
async function log(request, response, next) {
  response.write('Before response\n');
  await next();
  response.write('After response\n');

  // You cannot use response.end('After response\n')
  // because that would close the response stream
  // before Hello World! is written and the output
  // would be just:
  // Before response
  // After response
}

// Use authorization and logging middlewares
app.use(authorize, log);

app.get('/helloworld', (request, response) => {
  setTimeout(() => response.end('Hello World!\n'), 1000);
});
```

The reason is that the `next` function does not return a promise we could await. For this reason, the output from the */helloworld* endpoint would be in the wrong order:

```
Before response
After response
Hello World!
```

ESLint plugins utilize the chain of responsibility pattern, too. Below is code for defining one rule in an ESLint plugin:

```
create(context) {
  return {
    NewExpression(newExpr) {
      if (
        newExpr.callee.name === "SqlFilter" &&
        newExpr.arguments &&
        newExpr.arguments[0] &&
```

```
        newExpr.arguments[0].type !== "Literal"
      ) {
        context.report(
          newExpr,
          `SqlFilter constructor's 1st parameter must be a string literal`
        );
      }
    }
  };
}
```

ESLint plugin framework will call the `create` function and supply the `context` parameter. The `create` function should return an object of functions to analyze different *abstract syntax tree* (AST) nodes. In the above example, we are only interested in `NewExpression` nodes and analyze the creation of a new `SqlFilter` object. The first parameter supplied for the `SqlFilter` constructor should be a literal. If not, we report an issue using the `context.report` method.

When running ESLint with the above plugin and rule enabled, whenever ESLint encounters a `new` expression in a code file, the above-supplied `NewExpression` handler function will be called to check if the `new` expression in the code is valid.

The following code will pass the above ESLint rule:

```
const sqlFilter = new SqlFilter('field1 > 0');
```

And the following code won't:

```
const sqlExpression = 'field1 > 0';
const sqlFilter = new SqlFilter(sqlExpression);
```

Observer Pattern

> *The observer pattern lets you define an observe-notify (or publish-subscribe) mechanism to notify one or more objects about events that happen to the observed object.*

One typical example of using the observer pattern is a UI view observing a model. The UI view will be notified whenever the model changes and can redraw itself. Let's have an example with Java:

```
public interface Observer {
  void notifyAboutChange();
}

public interface Observable {
  void observeBy(Observer observer);
}

public class ObservableImpl {
  private final List<Observer> observers = new ArrayList<>();

  public void observeBy(final Observer observer) {
    observers.add(observer);
```

```
  }

  protected void notifyObservers() {
    observers.forEach(Observer::notifyAboutChange);
  }
}

public class TodosModel extends ObservableImpl {
  private List<Todo> todos = new ArrayList<>(25);

  // ...

  public void addTodo(final Todo todo) {
    todos.add(todo);
    notifyObservers();
  }

  public void removeTodo(final Todo todo) {
    todos.remove(todo);
    notifyObservers();
  }
}

public class TodosView implements Observer {
  private final TodosModel todosModel;

  public TodosView(final TodosModel todosModel) {
    this.todosModel = todosModel;
    todosModel.observeBy(this);
  }

  public void notifyAboutChange() {
    // Will be called when todos model change
    render();
  }

  public void render() {
    // Renders todos...
  }
}
```

Let's have another example that utilizes the publish-subscribe pattern. Below we define a `MessageBroker` class that contains the following methods: `publish`, `subscribe`, and `unsubscribe`.

```
nterface MessagePublisher<T> {
  void publish(String topic, T message);
}

@FunctionalInterface
interface MessageHandler<T> {
  void handle(T message);
}

interface MessageSubscriber<T> {
  void subscribe(String topic,
                 MessageHandler<T> messageHandler);
}

public class MessageBroker<T> implements
```

```
              MessagePublisher<T>, MessageSubscriber<T> {
  private final Map<String, List<MessageHandler<T>>>
    topicToMessageHandlersMap = new HashMap<>();

  public void publish(
    final String topic,
    final T message
  ) {
    final var messageHandlers =
      topicToMessageHandlersMap.get(topic);

    if (messageHandlers != null) {
      messageHandlers.forEach(messageHandler ->
        messageHandler.handle(message));
    }
  }

  public void subscribe(
    final String topic,
    final MessageHandler<T> messageHandler
  ) {
    final var messageHandlers =
      topicToMessageHandlersMap.get(topic);

      if (messageHandlers == null) {
        topicToMessageHandlersMap.put(topic,
                                      List.of(messageHandler));
      } else {
        messageHandlers.add(messageHandler);
      }
  }

   public void unsubscribe(
    final String topic,
    final MessageHandler<T> messageHandlerToRemove
  ) {
    final var messageHandlers =
      topicToMessageHandlersMap.get(topic);

    messageHandlers.removeIf(messageHandler ->
      messageHandler == messageHandlerToRemove);
  }
}
```

In the above example, we could have used the built-in Java `Consumer<T>` interface instead of the custom `MessageHandler<T>` interface.

Command/Action Pattern

> *Command or action pattern is used to define commands or actions as objects which can be given as parameters to other functions for later execution.*

Let's have an example using the Redux library well-known by many React developers. Below is a Redux reducer:

todosReducer.js

```
function todosReducer(state = initialState, action) {
  switch (action.type) {
    case 'ADD_TODO':
      return {
        ...state,
        todos: [...state.todos, {
          id: action.payload.id,
          name: action.payload.name,
          isDone: false
        }]
      };
    case 'MARK_TODO_DONE':
      const newTodos = state.todos.map(todo => {
        if (todo.id !== action.payload.id) {
          return todo;
        }

        return {
          ...todo,
          isDone: true
        };
      });

      return {
        ...state,
        todos: newTodos
      };
    default:
      return state;
  }
}
```

In the above example, we define a `todosReducer` which can handle two different actions: `ADD_TODO` and `MARK_TODO_DONE`. The implementation of the actions is inlined inside the switch statement, which makes the code somewhat hard to read. We can refactor the above code so that we introduce two classes for action objects:

AddTodoAction.ts

```
export default class AddTodoAction {
  constructor(
    private readonly id: number,
    private readonly name: string
  ) {}

  perform(state: TodoState): TodoState {
    return {
      ...state,
      todos: [...state.todos, {
        id: this.id,
        name: this.name,
        isDone: false
      }]
   };
  }
}
```

MarkDoneTodoAction.ts

```
export default class MarkDoneTodoAction {
  constructor(private readonly id: number) {}

  perform(state: TodoState): TodoState {
    const newTodos = state.todos.map(todo => {
      if (todo.id !== this.id) {
        return todo;
      }

      return {
        ...todo,
        isDone: true
      };
    });

    return {
      ...state,
      todos: newTodos
    };
  }
}
```

Now we can redesign the `todosReducer` to look like the following:

todosReducer.ts

```
import AddTodoAction from './AddTodoAction';
import MarkDoneTodoAction from './MarkDoneTodoAction';

function todosReducer(
  state: TodoState = initialState,
  { payload: { id, name }, type }: any
) {
  switch (type) {
    case 'ADD_TODO':
      return new AddTodoAction(id, name).perform(state);
    case 'MARK_TODO_DONE':
      return new MarkDoneTodoAction(id).perform(state);
    default:
      return state;
  }
}
```

We have separated actions into classes, and the `todosReducer` function becomes simpler. However, we should make the code object-oriented by replacing the conditionals (switch-case) with polymorphism. Let's do the following modifications: introduce a generic base class for actions and a base class for todo-related actions:

AbstractAction.ts

```
export default abstract class AbstractAction<S> {
  abstract perform(state: S): S;
}
```

AbstractTodoAction.ts

```
import AbstractAction from './AbstractAction';

export default abstract class AbstractTodoAction extends
  AbstractAction<TodoState> {}
```

The todo action classes must be modified to extend the `AbstractTodoAction` class:

AddTodoAction.ts

```
import AbstractTodoAction from './AbstractTodoAction';

export default class AddTodoAction extends AbstractTodoAction {
  // ...
}
```

MarkDoneTodoAction.ts

```
import AbstractTodoAction from './AbstractTodoAction';

export default class MarkDoneTodoAction extends AbstractTodoAction {
  // ...
}
```

Then we can introduce a generic function to create a reducer. This function will create a reducer function that perform actions for a given action base class:

createReducer.ts

```
import AbstractAction from './AbstractAction';

export default function createReducer<S>(
  initialState: S,
  ActionBaseClass:
    abstract new (...args: any[]) => AbstractAction<S>
) {
  return function(
    state: S = initialState,
    action: { type: AbstractAction<S> }
   ) {
     return action.type instanceof ActionBaseClass
       ? action.type.perform(state)
       : state;
  };
}
```

Let's create the initial state for todos:

Todo.ts

```
export type Todo = {
  id: number,
  name: string,
  isDone: boolean
}
```

initialTodosState.ts

```
import { Todo } from './Todo';

export type TodoState = {
  todos: Todo[];
}

const initialTodosState = {
  todos: []
} as TodoState

export default initialTodosState;
```

Next, we can create a Redux store using the `createReducer` function, the initial todo state, and the base action class for todo-related actions:

store.ts

```
import { combineReducers, createStore } from "redux";
import createReducer from "./createReducer";
import initialTodosState from "./initialTodoState";
import AbstractTodoAction from "./AbstractTodoAction";

const rootReducer = combineReducers({
  todoState: createReducer(initialTodosState, AbstractTodoAction)
});

export default createStore(rootReducer);
```

Now we have an object-oriented solution for dispatching actions in the following way:

```
dispatch({ type: new AddTodoAction(id, name) });
dispatch({ type: new MarkTodoDoneAction(id) });
```

Let's modify the `AbstractAction` class to support undoable actions. By default, an action is not undoable:

AbstractAction.ts

```
export default abstract class AbstractAction<S> {
  abstract perform(state: S): S;

  getName(): string {
    return this.constructor.name;
  }

  isUndoable(): boolean {
    return false;
  }
}
```

Let's also create a new class to serve as a base class for undoable actions:

AbstractUndoableAction.ts

```
import AbstractAction from "./AbstractAction";
```

```
export default abstract class AbstractUndoableAction<S> extends
         AbstractAction<S> {
  override isUndoable(): boolean {
    return true;
  }
}
```

Let's define a class for undo-actions. An undo-action sets the state as it was before performing the actual action.

UndoAction.ts

```
import AbstractAction from "./AbstractAction";

export default class UndoAction<S> extends AbstractAction<S> {
  constructor(
    private readonly actionName: string,
    private readonly ActionBaseClass:
      abstract new (...args: any[]) => AbstractAction<S>,
    private readonly state: S
  ) {
    super();
  }

  override getName(): string {
    return this.actionName;
  }

  override perform(state: S): S {
    return this.state;
  }

  getActionBaseClass():
    abstract new (...args: any[]) => AbstractAction<S>
  {
    return this.ActionBaseClass;
  }
}
```

Let's modify the `createReducer` function to create undo-actions for undoable actions and store them in a stack named `undoActions`. When a user wants to perform an undo of the last action, the topmost element from the `undoActions` stack can be popped and executed.

undoActions.ts

```
import UndoAction from "./UndoAction";

const undoActions = [] as UndoAction<any>[];
export default undoActions;
```

createReducer.ts

```
// ...
import undoActions from './undoActions';
import AbstractAction from "./AbstractAction";
import UndoAction from "./UndoAction";
```

```
function createReducer<S>(
  initialState: S,
  ActionBaseClass:
    abstract new (...args : any[]) => AbstractAction<S>
) {
  return function(
    state: S = initialState,
    action: { type: AbstractAction<S> }
  ) {
    let newState;

    if (action.type instanceof UndoAction &&
        action.type.getActionBaseClass() === ActionBaseClass) {
      newState = action.type.perform(state);
    } else if (action.type instanceof ActionBaseClass) {
      if (action.type.isUndoable()) {
          undoActions.unshift(new UndoAction(
            action.type.getName(),
            ActionBaseClass,
            state));
      }

      newState = action.type.perform(state);
    } else {
      newState = state;
    }

    return newState;
  };
}
```

Commands/Actions can also be defined without an object-oriented approach using a newly created function with a closure. In the below example, the function `() => toggleTodoDone(id)` is redefined for each todo. The function redefinition will always create a new closure that stores the current `id` variable value. We can treat the `() => toggleTodoDone(id)` as an action or command because it "encapsulates" the `id` value in the closure.

TodosTableView.tsx

```
// type Props = ...

export default function TodosTableView(
  { toggleTodoDone, todos }: Props
) {
  const todoElements = todos.map(({ id, name }) => (
    <tr>
      <td>{id}</td>
      <td>{name}</td>
      <td>
        <input
          type="checkbox"
          onChange={() => toggleTodoDone(id)}
        />
      </td>
    </tr>
  ));

  return <table><tbody>{todoElements}</tbody></table>;
}
```

Iterator Pattern

The iterator pattern can be used to add iteration capabilities to a sequence class.

Let's create a reverse iterator for Java's `List` class. We implement the `Iterator` interface by supplying implementations for the `hasNext` and the `next` methods:

ReverseListIterator.java

```
public class ReverseListIterator<T> implements Iterator<T> {
  private final List<T> values;
  private int iteratorPosition;

  public ReverseListIterator(final List<T> values) {
    this.values = Collections.unmodifiableList(values);
    iteratorPosition = values.size() - 1;
  }

  @Override
  public boolean hasNext() {
    return iteratorPosition >= 0;
  }

  @Override
  public T next() {
    // Note! We don't check the iteratorPosition
    // validity here, it is checked in hasNext() method,
    // which must be called before calling next() method
    // and only call next() method if hasNext() method
    // returned true
    final var nextValue = values.get(iteratorPosition);
    iteratorPosition--;
    return nextValue;
  }
}
```

We can put the `ReverseListIterator` class into use in a `ReverseArrayList` class defined below:

ReverseArrayList.java

```
public class ReverseArrayList<T> extends ArrayList<T>
{
  @Override
  public Iterator<T> iterator() {
    return new ReverseListIterator<>(this);
  }
}
```

Now we can use the new iterator to iterate over a list in reverse order:

```
final var reversedNumbers = new ReverseArrayList<Integer>();
reversedNumbers.addAll(List.of(1,2,3,4,5));

for (final var number : reversedNumbers) {
  System.out.println(number);
}
```

```
// Prints:
// 5
// 4
// 3
// 2
// 1
```

State Pattern

The state pattern lets an object change its behavior depending on its current state.

Developers don't often treat an object's state as an object but as an enumerated value (enum), for example. Below is an example where we have defined a `UserStory` class representing a user story that can be rendered on screen. An enum value represents the state of a `UserStory` object.

```
public enum UserStoryState {
  TODO, IN_DEVELOPMENT, IN_VERIFICATION, READY_FOR_REVIEW, DONE
}

public class UserStory {
  private String name;
  private UserStoryState state = UserStoryState.TODO;
  // Other properties...

  public UserStory(final String name, ...) {
    this.name = name;
    // ...
  }

  public void setState(
    final UserStoryState newState
  ) {
    state = newState;
  }

  public void render() {
    final var icon = switch(state) {
      case TODO -> new TodoIcon();
      case IN_DEVELOPMENT -> new InDevelopmentIcon();
      case IN_VERIFICATION -> new InVerificationIcon();
      case READY_FOR_REVIEW -> new ReadyForReviewIcon();
      case DONE -> new DoneIcon();
      default -> throw new IllegalArgumentException(...);
    };

    // Draw a UI elements on screen representing the user story
    // using the given 'icon'
  }
}
```

The above solution is not an object-oriented one. We should replace the conditionals (switch-case statement) with a polymorphic design. This can be done by introducing state objects. In the state pattern, the state of an object is represented with an object instead of an enum value. Below is the above code modified to use the state pattern:

```
public interface UserStoryState {
```

```
  Icon getIcon();
}

public class TodoUserStoryState implements UserStoryState {
  public Icon getIcon() {
    return new TodoIcon();
  }
}

public class InDevelopmentUserStoryState
         implements UserStoryState {
  public Icon getIcon() {
    return new InDevelopmentIcon();
  }
}

public class InVerificationUserStoryState
         implements UserStoryState {
  public Icon getIcon() {
    return new InVerificationIcon();
  }
}

public class ReadyForReviewUserStoryState
         implements UserStoryState {
  public Icon getIcon() {
    return new ReadyForReviewIcon();
  }
}

public class DoneUserStoryState
         implements UserStoryState {
  public Icon getIcon() {
    return new DoneIcon();
  }
}

public class UserStory {
  private String name;
  private UserStoryState state = new TodoUserStoryState();
  // Other properties...

  public UserStory(final String name, ...) {
    this.name = name;
    // ...
  }

  public void setState(
    final UserStoryState newState
  ) {
    state = newState;
  }

  public void render() {
    final Icon icon = state.getIcon();
    // Draw a UI element on screen representing
    // the user story using the given 'icon'
  }
}
```

Let's have another example with an `Order` class. An order can have a state, like paid, packaged,

delivered, etc. Below we implement the order states as classes:

```
public interface OrderState {
  String getMessage(String orderId);
}

public class PaidOrderState implements OrderState {
  public String getMessage(final String orderId) {
    return "Order " + orderId + " is successfully paid";
  }
}

public class DeliveredOrderState implements OrderState {
  public String getMessage(final String orderId) {
    return "Order " + orderId + " is delivered";
  }
}

// Implement the rest of possible order states here...

public class Order {
  private String id;
  private OrderState state;
  private Customer customer;

  // ...

  public String getCustomerEmailAddress() {
    return customer.getEmailAddress();
  }

  public String getStateMessage() {
    return state.getMessage(id);
  }
}

emailService.sendEmail(order.getCustomerEmailAddress(),
                       order.getStateMessage());
```

Mediator Pattern

> *The mediator pattern lets you reduce dependencies between objects. It restricts direct communication between two different layers of objects and forces them to collaborate only via a mediator object or objects.*

The mediator pattern eliminates the coupling of two different layers of objects. So changes to one layer of objects can be made without the need to change the objects in the other layer.

A typical example of the mediator pattern is Model-View-Controller (MVC) pattern. In the MVC pattern, model and view objects do not communicate directly but only via mediator objects (controllers). Next, several different ways to use the MVC pattern in frontend clients are presented. Traditionally MVC pattern was used in the backend when the backend also generated the view to be shown in the client device (web browser). With the advent of single-page web clients, a modern backend is a simple API containing only a model and controller (MC).

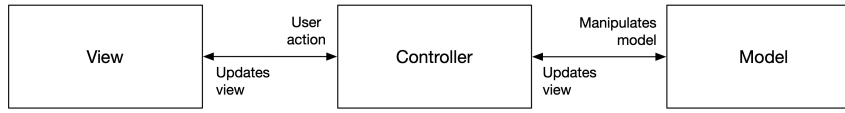

Fig 3.6 Model-View-Controller

In the below picture, you can see how dependency inversion is used, and none of the implementation classes depend on concrete implementations. You can easily change any implementation class to a different one without the need to modify any other implementation class. Notice how the `ControllerImpl` class uses the *bridge pattern* and implements two bridges, one towards the model and the other towards the view.

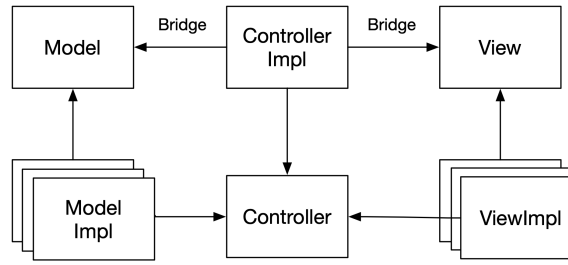

Fig 3.7 Dependencies in MVC pattern

As shown in the below picture, the controller can also be used as a bridge-adapter: The controller can be modified to adapt to changes in the view layer (`View2` instead of `View`) without needing to change the model layer. The modified modules are shown with a grey background in the picture. Similarly, the controller can be modified to adapt to changes in the model layer without needing to change the view layer (not shown in the picture).

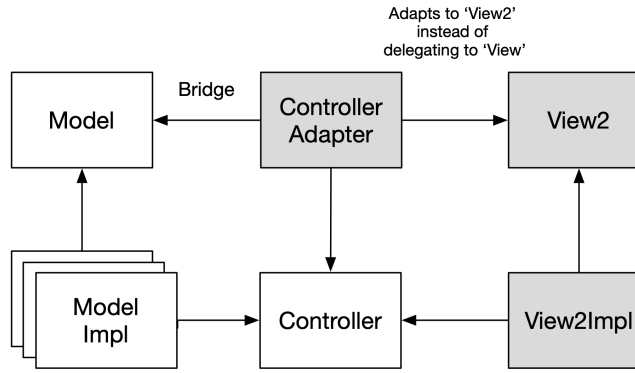

Fig 3.8 Adapting to Changes in MVC pattern

The following examples use a specialization of the MVC pattern called Model-View-Presenter (MVP). In the MVP pattern, the controller is called the presenter. I use the more generic term *controller* in all examples, though. A Presenter act as a middle-man between a view and a model. A presenter-type controller object has a reference to a view object and a model object. A view object commands the presenter to perform actions on the model. And the model object asks the presenter to update the view object.

In the past, making desktop UI applications using Java Swing as the UI layer was popular. Let's have a simple todo application as an example:

First, we implement the `Todo` class, which is part of the model.

Todo.java

```
public class Todo {
  private int id;
  private String name;
  private boolean isDone;

  // Constructor...

  public int getId() {
    return id;
  }

  public void setId(final int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(final String name) {
    this.name = name;
  }

  public boolean isDone() {
    return isDone;
  }

  public void setIsDone(final boolean isDone) {
    this.isDone = isDone;
  }
}
```

Next, we implement the view layer:

TodoView.java

```
public interface TodoView {
  void show(List<Todo> todos);
  void show(String message);
}
```

TodoViewImpl.java

```
public class TodoViewImpl implements TodoView {
  private final TodoController controller;

  public TodoViewImpl(final TodoController controller) {
    this.controller = controller;
    controller.setView(this);
    controller.startFetchTodos();
  }

  public void show(final List<Todo> todos) {
    // Update the view to show the given todos
    // Add listener for each todo checkbox.
    // Listener should call: controller.toggleTodoDone(todo.id)
  }

  public void show(final String errorMessage) {
    // Update the view to show error message
  }
}
```

Then we implement a generic `Controller` class that acts as a base class for concrete controllers:

Controller.java

```
public class Controller<M, V> {
  private M model;
  private V view;

  public M getModel() {
    return model;
  }

  public void setModel(final M model) {
    this.model = model;
  }

  public V getView() {
    return view;
  }

  public void setView(final V view) {
    this.view = view;
  }
}
```

The below `TodoControllerImpl` class implements two actions, `startFetchTodos` and `toggleTodoDone`, which delegate to the model layer. It also implements two actions, `updateViewWith(todos)` and `updateViewWith(errorMessage)`, that delegate to the view layer. The latter two actions are executed in the Swing UI thread using `SwingUtilities.invokeLater`.

TodoController.java

```
public interface TodoController {
  void startFetchTodos();
  void toggleTodoDone(final int id);
  void updateViewWith(final List<Todo> todos);
  void updateViewWith(final String errorMessage);
```

```
}
```

TodoControllerImpl.java

```
public class TodoControllerImpl
         extends Controller<TodoModel, TodoView>
         implements TodoController {

  public void startFetchTodos() {
    getModel().fetchTodos();
  }

  public void toggleTodoDone(final int id) {
    getModel().toggleTodoDone(id);
  }

  public void updateViewWith(final List<Todo> todos) {
    SwingUtilities.invokeLater(() ->
      getView().show(todos));
  }

  public void updateViewWith(final String errorMessage) {
    SwingUtilities.invokeLater(() ->
      getView().show(errorMessage));
  }
}
```

The below `TodoModelImpl` class implements the fetching of todos (`fetchTodos`) using the supplied `todoService`. The `todoService` accesses the backend to read todos from a database, for example. When todos are successfully fetched, the controller is told to update the view. If fetching of the todos fails, the view is updated to show an error. Toggling a todo done is implemented using the `todoService` and its `updateTodo` method.

TodoService.java

```
public interface TodoService {
  public List<Todo> getTodos();
  public void updateTodo(Todo todo);
}
```

TodoModel.java

```
public interface TodoModel {
  public void fetchTodos();
  public void toggleTodoDone(int id);
}
```

TodoModelImpl.java

```
public class TodoModelImpl implements TodoModel {
  private final TodoController controller;
  private final TodoService todoService;
  private List<Todo> todos = new ArrayList<>();

  public TodoModelImpl(
    final TodoController controller,
    final TodoService todoService
  ) {
```

```
    this.controller = controller;
    controller.setModel(this);
    this.todoService = todoService;
  }

  public void fetchTodos() {
    CompletableFuture
      .supplyAsync(todoService::getTodos)
      .thenAccept(todos -> {
        this.todos = todos;
        controller.updateViewWith(todos);
      })
      .exceptionally((error) -> {
        controller.updateViewWith(error.getMessage());
        return null;
      });
  }

  public void toggleTodoDone(final int id) {
    todos.stream()
      .filter(todo -> todo.getId() == id)
      .findAny()
      .ifPresent(todo -> {
          todo.setIsDone(!todo.isDone());

          CompletableFuture
            .runAsync(() ->
              todoService.updateTodo(todo))
            .exceptionally((error) -> {
              controller.updateViewWith(error.getMessage());
              return null;
            });
      });
  }
}
```

Let's have the same example using *Web Components*. The web component view should extend the `HTMLElement` class. The `connectedCallback` method of the view will be called on the component mount. It starts fetching todos. The `showTodos` method renders the given todos as HTML elements. It also adds event listeners for the *Mark done* buttons. The `showError` method updates the inner HTML of the view to show an error message.

Todo.ts

```
export type Todo = {
  id: number;
  name: string;
  isDone: boolean;
};
```

TodoView.ts

```
interface TodoView {
  showTodos(todos: Todo[]): void;
  showError(errorMessage: string): void;
}
```

TodoViewImpl.ts

```
import controller from './todoController';
import { Todo } from './Todo';

export default class TodoViewImpl
         extends HTMLElement implements TodoView {
  constructor() {
    super();
    controller.setView(this);
  }

  connectedCallback() {
    controller.startFetchTodos();
    this.innerHTML = '<div>Loading todos...</div>';
  }

  showTodos(todos: Todo[]) {
    const todoElements = todos.map(({ id, name, isDone }) => `
      <li id="todo-${id}">
        ${id} ${name} 
        ${isDone ? '' : '<button>Mark done</button>'}
      </li>
    `);

    this.innerHTML = `<ul>${todoElements}</ul>`;

    todos.map(({ id }) => this
      .querySelector(`#todo-${id} button`)?
      .addEventListener('click',
                        () => controller.toggleTodoDone(id)));
  }

  showError(errorMessage: string) {
    this.innerHTML = `
      <div>
        Failure: ${errorMessage}
      </div>
    `;
  }
}
```

We can use the same controller and model APIs for this web component example as in the Java Swing example. We just need to convert the Java code to TypeScript code:

Controller.ts

```
export default class Controller<M, V> {
  private model: M | undefined;
  private view: V | undefined;

  getModel(): M | undefined {
    return this.model;
  }

  setModel(model: M): void {
    this.model = model;
  }

  getView(): V | undefined {
    return this.view;
  }
```

```
  setView(view: V): void {
    this.view = view;
  }
}
```

TodoController.ts

```
import { Todo } from "./Todo";

export interface TodoController {
  startFetchTodos(): void;
  toggleTodoDone(id: number): void;
  updateViewWithTodos(todos: Todo[]): void;
  updateViewWithError(message: string): void;
}
```

todoController.ts

```
import TodoView from './TodoView';
import Controller from "./Controller";
import { TodoController } from './TodoController';
import { Todo } from "./Todo";
import TodoModel from './TodoModel';

class TodoControllerImpl
         extends Controller<TodoModel, TodoView>
         implements TodoController {

  startFetchTodos(): void {
    this.getModel()?.fetchTodos();
  }

  toggleTodoDone(id: number): void {
    this.getModel()?.toggleTodoDone(id);
  }

  updateViewWithTodos(todos: Todo[]): void {
    this.getView()?.showTodos(todos);
  }

  updateViewWithError(message: string): void {
    this.getView()?.showError(message);
  }
}

const controller = new TodoControllerImpl();
export default controller;
```

TodoService.ts

```
export interface TodoService {
  getTodos(): Promise<Todo[]>;
  updateTodo(todo: Todo): Promise<void>;
}
```

TodoModel.ts

```
export interface TodoModel {
  fetchTodos(): void;
  toggleTodoDone(id: number): void;
}
```

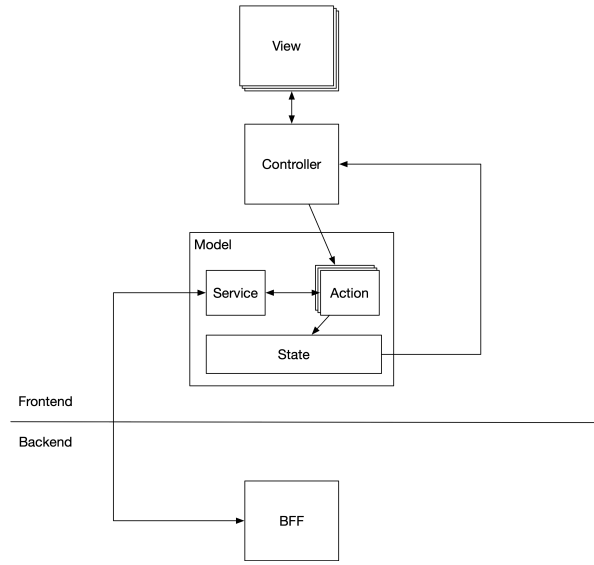

Figure 3.9 Frontend MVC Architecture with Redux

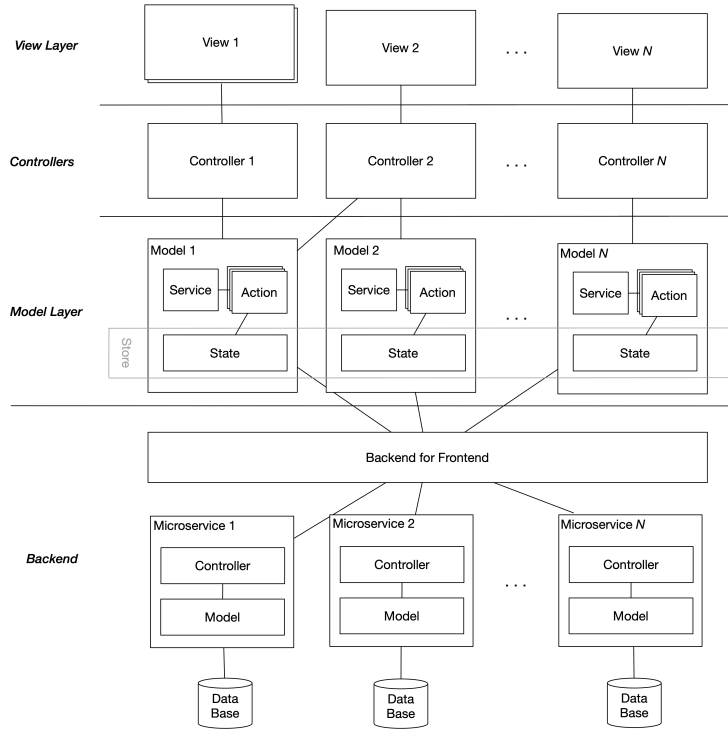

Figure 3.10 Frontend MVC Architecture with Redux + Backend

TodoModelImpl.ts

```
import controller, { TodoController } from './todoController';
import { TodoModel } from './TodoModel';
import { Todo } from "./Todo";

export default class TodoModelImpl implements TodoModel {
  private todos: Todo[] = [];

  constructor(
    private readonly controller: TodoController,
    private readonly todoService: TodoService
  ) {
    controller.setModel(this);
  }

  fetchTodos(): void {
    this.todoService.getTodos()
      .then((todos) => {
        this.todos = todos;
        controller.updateViewWithTodos(todos);
      })
      .catch((error) =>
        controller.updateViewWithError(error.message));
  }

  toggleTodoDone(id: number): void {
    const foundTodo = this.todos.find(todo => todo.id === id);

    if (foundTodo) {
      foundTodo.isDone = !foundTodo.isDone;
      this.todoService
          .updateTodo(foundTodo)
          .catch((error: any) =>
            controller.updateViewWithError(error.message));
    }
  }
}
```

We could use the above-defined controller and model as such with a React view component:

ReactTodoView.tsx

```
// ...
import controller from './todoController';

// ...

export default class ReactTodoView
        extends Component<Props, State>
        implements TodoView {

  constructor(props: Props) {
    super(props);
    controller.setView(this);

    this.state = {
      todos: []
    }
  }
```

```
  componentDidMount() {
    controller.startFetchTodos();
  }

  showTodos(todos: Todo[]) {
    this.setState({ ...this.state, todos });
  }

  showError(errorMessage: string) {
     this.setState({ ...this.state, errorMessage });
  }

  render() {
    // Render todos from 'this.state.todos' here
    // Or show 'this.state.errorMessage' here
  }
}
```

If you have multiple views using the same controller, you can derive your controller from the below-defined `MultiViewController` class:

MultiViewController.ts

```
export default class MultiViewController<M, V> {
  private model: M | undefined;
  private views: V[] = [];

  getModel(): M | undefined {
    return this.model;
  }

  setModel(model: M): void {
    this.model = model;
  }

  getViews(): V[]  {
    return this.views;
  }

  addView(view: V): void {
    this.views.push(view);
  }
}
```

Let's say we want to have two views for todos, one for the actual todos and one viewing the todo count. We need to modify the controller slightly to support multiple views:

todoController.ts

```
import TodoView from './TodoView';
import MultiViewController from './MultiViewController';
import { Todo } from "./Todo";
import { TodoController } from './TodoController';
import TodoModel from './TodoModel';

class TodoControllerImpl
         extends MultiViewController<TodoModel, TodoView>
         implement TodoController {
  startFetchTodos(): void {
```

```
    this.getModel()?.fetchTodos();
  }

  toggleTodoDone(id: number): void {
    this.getModel()?.toggleTodoDone(id);
  }

  updateViewsWithTodos(todos: Todo[]): void {
    this.getViews().forEach(view => view.showTodos(todos));
  }

  updateViewWithError(message: string): void {
    this.getViews().forEach(view => view.showError(message));
  }
}

const controller = new TodoController();
export default controller;
```

Many modern UI frameworks and state management libraries implement a specialization of the MVC pattern called, Model-View-ViewModel (MVVM). In the MVVM pattern, the controller is called the view model. I use the more generic term *controller* in the below example, though. The main difference between the view model and the presenter in the MVP pattern is that in the MVP pattern, the presenter has a reference to the view, but the view model does not. The view model provides bindings between the view's events and actions in the model. This can happen so that the view model adds action dispatcher functions as properties of the view. And in the other direction, the view model maps the model's state to the properties of the view. When using React and Redux, for example, you can connect the view to the model using the `mapDispatchToProps` function and connect the model to the view using the `mapStateToProps` function. These two mapping functions form the view model (or the controller) that binds the view and model together.

Let's first implement the todo example with React and Redux and later show how the React view can be replaced with an Angular view without any modification to the controller or the model layer. Note that the code for some classes is not listed below. You can assume those classes are the same as defined in *command/action pattern* examples.

Let's implement a list view for todos:

TodosListView.tsx

```
import { connect } from 'react-redux';
import { useEffect } from "react";
import { controller, ActionDispatchers, State }
  from './todosController';

type Props = ActionDispatchers & State;

function TodosListView({
  toggleTodoDone,
  startFetchTodos,
  todos
}: Props) {

  useEffect(() => {
```

```
    startFetchTodos();
  }, [startFetchTodos]);

  const todoElements = todos.map(({ id, name, isDone }) => (
    <li key={id}>
      {id} 
      {name} 
      {isDone
        ? undefined
        : <button onClick={() => toggleTodoDone(id)}>
            Mark done
          </button>
      }
    </li>
  ));

  return <ul>{todoElements}</ul>;
}

// Here we connect the view to the model using the controller
export default connect(
  controller.getState,
  () => controller.getActionDispatchers()
)(TodosListView);
```

Below is the base class for controllers:

Controller.ts

```
import AbstractAction from "./AbstractAction";

export type Dispatch =
 (plainActionObject: { type: AbstractAction<any> }) => void;

export default class Controller {
  protected readonly dispatch:
    (action: AbstractAction<any>) => void;

  // The 'dispatch' is from Redux library
  constructor(dispatch: Dispatch) {
    this.dispatch = (action: AbstractAction<any>) =>
      dispatch({ type: action });
  }
}
```

Below is the controller for todos:

todosController.ts

```
import store from './store';
import { AppState } from "./AppState";
import ToggleDoneTodoAction from "./ToggleDoneTodoAction";
import StartFetchTodosAction from "./StartFetchTodosAction";
import Controller from "./Controller";

class TodosController extends Controller {
  getState(appState: AppState) {
    return {
      todos: appState.todosState.todos,
    }
  }
```

```
  getActionDispatchers() {
    return {
      toggleTodoDone: (id: number) =>
        this.dispatch(new ToggleDoneTodoAction(id)),

      startFetchTodos: () =>
        this.dispatch(new StartFetchTodosAction())
    }
  }
}

export const controller = new TodosController(store.dispatch);
export type State = ReturnType<typeof controller.getState>;

export type ActionDispatchers =
  ReturnType<typeof controller.getActionDispatchers>;
```

In the development phase, we can use the following temporary implementation of the `StartFetchTodosAction` class:

StartFetchTodosAction.ts

```
import { TodoState } from "./TodoState";
import AbstractTodoAction from "./AbstractTodoAction";

export default class StartFetchTodosAction extends
                       AbstractTodoAction {
  perform(state: TodoState): TodoState {
    return {
      todos: [
        {
          id: 1,
          name: "Todo 1",
          isDone: false,
        },
        {
          id: 2,
          name: "Todo 2",
          isDone: false,
        },
      ],
    };
  }
}
```

Now we can introduce a new view for todos, a `TodosTableView` which can utilize the same controller as the `TodosListView`.

TodosTableView.tsx

```
import { connect } from 'react-redux';
import { useEffect } from "react";
import { controller, ActionDispatchers, State }
  from './todosController';

type Props = ActionDispatchers & State;

function TodosListView({
  toggleTodoDone,
```

```
  startFetchTodos,
  todos
}: Props) {
  useEffect(() => {
    startFetchTodos();
  }, [startFetchTodos]);

  const todoElements = todos.map(({ id, isDone, name }) => (
    <tr key={id}>
      <td>{id}</td>
      <td>{name}</td>
        <td>
          <input
            type="checkbox"
            checked={isDone}
            onChange={() => toggleTodoDone(id)}
          />
        </td>
    </tr>
  ));

  return <table><tbody>{todoElements}</tbody></table>;
}

export default connect(
  controller.getState,
  () => controller.getActionDispatchers()
)(TodosListView);
```

We can notice some duplication in the `TodosListView` and `TodosTableView` components. For example, both are using the same effect. We can create a `TodosView` for which we can give as parameter the type of a single todo view, either a list item or a table row view:

TodosView.tsx

```
import { useEffect } from "react";
import { connect } from "react-redux";
import ListItemTodoView from './ListItemTodoView';
import TableRowTodoView from './TableRowTodoView';
import { controller, ActionDispatchers, State }
  from './todosController';

type Props = ActionDispatchers & State & {
    TodoView: typeof ListItemTodoView | typeof TableRowTodoView;
};

function TodosView({
  toggleTodoDone,
  startFetchTodos,
  todos,
  TodoView
}: Props) {
  useEffect(() => {
    startFetchTodos()
  }, [startFetchTodos]);

  const todoViews = todos.map((todo) =>
    <TodoView
      key={todo.id}
      todo={todo}
      toggleTodoDone={toggleTodoDone}
```

```
    />
  );

  return TodoView === ListItemTodoView
      ? <ul>{todoViews}</ul>
      : <table><tbody>{todoViews}</tbody></table>;
}

export default connect(
  controller.getState,
  () => controller.getActionDispatchers()
)(TodosView);
```

Below is the view for showing a single todo as a list item:

TodoViewProps.ts

```
import { Todo } from "./Todo";

export type TodoViewProps = {
  toggleTodoDone: (id: number) => void,
  todo: Todo
}
```

ListItemTodoView.tsx

```
import { TodoViewProps } from './TodoViewProps';

export default function ListItemTodoView({
  toggleTodoDone,
  todo: { id, name, isDone }
}: TodoViewProps) {
  return (
    <li>
      {id} 
      {name} 
      { isDone ?
        undefined :
        <button onClick={() => toggleTodoDone(id)}>
          Mark done
        </button> }
    </li>
  );
}
```

Below is the view for showing a single todo as a table row:

TableRowTodoView.tsx

```
import { TodoViewProps } from './TodoViewProps';

export default function TableRowTodoView({
  toggleTodoDone,
  todo: { id, name, isDone }
}: TodoViewProps) {
  return (
    <tr>
      <td>{id}</td>
      <td>{name}</td>
      <td>
```

```
        <input
          type="checkbox"
          checked={isDone}
          onChange={() => toggleTodoDone(id)}
        />
      </td>
    </tr>);
}
```

In most cases, you should not store state in a view even if the state is for that particular view only. Instead, when you store it in the model, it brings the following benefits:

- Possibility to easily persist state either in the browser or in the backend
- Possibility to easily implement undo-actions
- State can be easily shared with another view(s) later if needed
- Migrating views to use a different view technology is more straightforward
- Easier debugging of state-related problems, e.g., using the Redux DevTools browser extension

We can also change the view implementation from React to Angular without modifying the controller or model layer. This can be done, for example, using the *@angular-redux2/store* library. Below is a todos table view implemented as an Angular component:

todos-table-view.component.ts

```
import { Component, OnInit } from "@angular/core";
import { NgRedux, Select } from '@angular-redux2/store';
import { Observable } from "rxjs";
import { controller } from './todosController';
import { TodoState } from "./TodoState";
import { AppState } from "./AppState";

const { startFetchTodos,
        toggleTodoDone } = controller.getActionDispatchers();

@Component({
  selector: 'todos-table-view',
  template: `
    <table>
      <tr *ngFor="let todo of (todoState | async)?.todos">
        <td>{{ todo.id }}</td>
        <td>{{ todo.name }}</td>
        <td>
          <input
            type="checkbox"
            [checked]="todo.isDone"
            (change)="toggleTodoDone(todo.id)"
          />
        </td>
      </tr>
    </table>
  `
})
export class TodosTableView implements OnInit {
  @Select(controller.getState) todoState: Observable<TodoState>;

  constructor(private ngRedux: NgRedux<AppState>) {}
```

```
  ngOnInit(): void {
    startFetchTodos();
  }

  toggleTodoDone(id: number) {
    toggleTodoDone(id);
  }
}
```

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
  <div>
    <todos-table-view></todos-table-view>
  </div>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-test';
}
```

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { NgReduxModule, NgRedux } from '@angular-redux2/store';

import { AppComponent } from './app.component';
import store from './store';
import { AppState } from "./AppState";
import { TodosTableView } from "./todos-table-view.component";

@NgModule({
  declarations: [
    AppComponent, TodosTableView
  ],
  imports: [
    BrowserModule,
    NgReduxModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
  constructor(ngRedux: NgRedux<AppState>) {
    ngRedux.provideStore(store);
  }
}
```

Template Method Pattern

> *Template method pattern allows you to define a template method in a base class, and subclasses define the final implementation of that method. The template method contains one or more calls to abstract methods implemented in the subclasses.*

In the below example, the `AbstractDrawing` class contains a template method, `draw`. This method includes a call to the `getShapeRenderer` method, an abstract method implemented in the subclasses of the `AbstractDrawing` class. The `draw` method is a template method, and a subclass defines how to draw a single shape.

```
public interface Drawing {
  ShapeRenderer getShapeRenderer();
  void draw();
}

public abstract class AbstractDrawing implements Drawing {
  private final List<Shape> shapes;

  public AbstractDrawing(final List<Shape> shapes) {
    this.shapes = shapes;
  }

  public abstract ShapeRenderer getShapeRenderer();

  public final void draw() {
    for (final Shape shape: shapes) {
      shape.render(getShapeRenderer());
    }
  }
}
```

We can now implement two subclasses of the `AbstractDrawing` class, which define the final behavior of the templated `draw` method. We mark the template method `draw` as `final` because subclasses should not override it. It is best practice to declare a template method as final. They should only provide an implementation for the abstract `getShapeRenderer` method.

```
public class RasterDrawing extends AbstractDrawing {
  private final Canvas canvas = new Canvas();

  private final ShapeRenderer shapeRendrerer =
    new RasterShapeRenderer(canvas);

  public RasterDrawing(final List<Shape> shapes) {
    super(shapes);
  }

  public ShapeRenderer getShapeRenderer() {
    return shapeRenderer;
  }
}

public class VectorDrawing extends AbstractDrawing {
  private final SVGElement svgRoot = new SVGElement();

  private final ShapeRenderer shapeRenderer =
    new VectorShapeRenderer(svgRoot);

  public VectorDrawing(final List<Shape> shapes) {
    super(shapes);
  }

  public ShapeRenderer getShapeRenderer() {
```

```
    return shapeRenderer;
  }
}
```

Memento Pattern

> *The memento pattern can be used to save the internal state of an object to another object called the memento object.*

Let's have an example with a `TextEditor` class. First, we define a `TextEditorState` interface and its implementation. Then we define a `TextEditorStateMemento` class for storing a memento of the text editor's state.

```
public interface TextEditorState {
  TextEditorState clone();
}

public class TextEditorStateImpl implements TextEditorState {
  // Implement text editor state here
}

public class TextEditorStateMemento {
  private final TextEditorState state;

  public TextEditorStateMemento(final TextEditorState state) {
    this.state = state.clone();
  }

  public TextEditorState getState() {
    return state;
  }
}
```

The `TextEditor` class stores mementos of the text editor's state. It provides methods to save a state, restore a state, or restore the previous state:

TextEditor.java

```
class TextEditor {
  private final List<TextEditorStateMemento> stateMementos =
    new ArrayList<>(20);

  private TextEditorState currentState;
  private int currentVersion = 1;

  public void saveState() {
    stateMementos.add(new TextEditorStateMemento(currentState));
    currentVersion += 1;
  }

  public void restoreState(final int version) {
    if (version >= 1 && version <= stateMementos.size()) {
      currentState = stateMementos.get(version - 1).getState();
      currentVersion += 1;
    }
  }
```

```
  public void restorePreviousState() {
    if (currentVersion > 1) {
      restoreState(currentVersion - 1);
    }
  }
}
```

In the above example, we can add a memento for the text editor's state by calling the `saveState` method. We can recall the previous version of the text editor's state with the `restorePreviousState` method, and we can recall any version of the text editor's state using the `restoreState` method.

Visitor Pattern

> *Visitor pattern allows adding functionality to a class (like adding new methods) without modifying the class. This is useful, for example, with library classes that you cannot modify.*

First, let's have an example with classes that we can modify:

```
public interface Shape {
  void draw();
}

public class CircleShape implements Shape {
  private final int radius;

  // ...

  public void draw() {
    // ...
  }

  public int getRadius() {
    return radius;
  }
}

public class RectangleShape implements Shape {
  private final int width;
  private final int height;

  // ...

  public void draw() {
    // ...
  }

  public int getWidth() {
    return width;
  }

  public int getHeight() {
    return height;
  }
}
```

Let's assume we need to calculate the total area of shapes in a drawing. Currently, we are in a situation where we can modify the shape classes, so let's add `calculateArea` methods to the classes:

```
public interface Shape {
  // ...

  double calculateArea();
}

public class CircleShape implements Shape {
  // ...

  public double calculateArea() {
    return Math.PI * radius * radius;
  }
}

class RectangleShape implements Shape {
  // ...

  public double calculateArea() {
    return width * height;
  }
}
```

Adding a new method to an existing class may be against the *open-closed principle*. In the above case, adding the `calculateArea` methods is safe because the shape classes are immutable. And even if they were not, adding the `calculateArea` methods would be safe because they are read-only methods, i.e., they don't modify the object's state, and we don't have to worry about thread safety because we can agree that our example application is not multithreaded.

Now we have the area calculation methods added, and we can use a common algorithm to calculate the total area of shapes in a drawing:

```
final var totalAreaOfShapes = drawing
  .getShapes()
  .stream()
  .reduce(0.0, (subTotalArea, shape) ->
     subTotalArea + shape.calculateArea(), Double::sum);
```

But what if the shape classes, without the area calculation capability, were in a 3rd party library that we cannot modify? We would have to do something like this:

```
final var totalAreaOfShapes = drawing
  .getShapes()
  .stream()
  .reduce(0.0, (subTotalArea, shape) -> {
    double shapeArea;

    if (shape instanceof CircleShape) {
      shapeArea = Math.PI *
                  Math.pow(((CircleShape)shape).getRadius(), 2);
    } else if (shape instanceof RectangleShape){
      shapeArea = ((RectangleShape)shape).getWidth() *
```

```
                  ((RectangleShape)shape).getHeight();
    }
    else {
      throw new IllegalArgumentException("Invalid shape");
    }

    return subTotalArea + shapeArea;
  }, Double::sum);
```

The above solution is complicated and needs updating every time a new type of shape is introduced. The above example does not follow object-oriented design principles: it contains an if/else-if structure with `instanceof` checks.

We can use the visitor pattern to replace the above conditionals with polymorphism. First, we introduce a visitor interface that can be used to provide additional behavior to the shape classes. Then we introduce an `execute` method in the `Shape` interface. And in the shape classes, we implement the `execute` methods so that additional behavior provided by a concrete visitor can be executed:

```
// This is our visitor interface that
// provides additional behaviour to the shape classes
public interface ShapeBehavior {
  Object executeForCircle(final CircleShape circle);
  Object executeForRectangle(final RectangleShape rectangle);

  // Add methods for possible other shape classes here...
}

public interface Shape {
  // ...

  Object execute(final ShapeBehavior behavior);
}

public class CircleShape implements Shape {
  public Object execute(final ShapeBehavior behavior) {
    return behavior.executeForCircle(this);
  }
}

public class RectangleShape implements Shape {
  public Object execute(final ShapeBehavior behavior) {
    return behavior.executeForRectangle(this);
  }
}
```

Suppose that the shape classes were mutable and made thread-safe. We would have to define the `execute` methods with appropriate synchronization to make them also thread-safe:

```
public class CircleShape implements Shape {
  public synchronized Object execute(
    final ShapeBehavior behavior
  ) {
    return behavior.executeForCircle(this);
  }
}
```

```
public class RectangleShape implements Shape {
  public synchronized Object execute(
    final ShapeBehavior behavior
  ) {
    return behavior.executeForRectangle(this);
  }
}
```

Let's implement a concrete visitor for calculating areas of different shapes:

```
public class AreaCalculationShapeBehavior implements
               ShapeBehavior {
  public Object executeForCircle(final CircleShape circle) {
    return (Double)(Math.PI *
                    Math.pow(circle.getRadius(), 2));
  }

  public Object executeForRectangle(
    final RectangleShape rectangle
  ) {
    return (Double)(double)(rectangle.getWidth() *
                            rectangle.getHeight());
  }
}
```

Now we can implement the calculation of shapes' total area using a common algorithm, and we get rid of the conditionals. We execute the `areaCalculation` behavior for each shape and convert the result of behavior execution to `Double`. Methods in a visitor usually return some common type like `Object`. This enables various operations to be performed. After executing a visitor, the return value should be cast to the right type.

```
final var areaCalculation = new AreaCalculationShapeBehavior();

final var totalAreaOfShapes = drawing
  .getShapes()
  .stream()
  .reduce(0.0, (subTotalArea, shape) ->
    subTotalArea + (Double)shape.execute(areaCalculation),
      Double::sum);
```

You can add more behavior to the shape classes by defining a new visitor. Let's define a `PerimeterCalculationShapeBehaviour` class:

```
public class PerimeterCalculationShapeBehavior
               implements ShapeBehavior {
  public Object executeForCircle(final CircleShape circle) {
    return (Double)(2 * Math.PI * circle.getRadius());
  }

  public Object executeForRectangle(
    final RectangleShape rectangle
  ) {
    return (Double)(double)(2 * rectangle.getWidth() +
                            2 * rectangle.getHeight());
```

```
  }
}
```

Notice that we did not need to use the *visitor* term in our code examples. Adding the design pattern name to the names of software entities (class/function names, etc.) often does not bring any real benefit but makes the names longer. However, there are some design patterns, like the *factory pattern* and *builder pattern* where you always use the design pattern name in a class name.

If you develop a third-party library and would like the behavior of its classes to be extended by its users, you should make your library classes accept visitors that can perform additional behavior.

Null Object Pattern

> *A null object is an object that does nothing.*

Use the null object pattern to implement a class for null objects that don't do anything. A null object can be used in place of a real object that does something.

Let's have an example with a `Shape` interface:

Shape.java

```
public interface Shape {
  void draw();
}
```

We can easily define a class for null shape objects:

NullShape.java

```
public class NullShape implements Shape {
  void draw() {
    // Intentionally no operation
  }
}
```

We can use an instance of the `NullShape` class everywhere where a concrete implementation of the `Shape` interface is wanted.

Don't Ask, Tell Principle

> *Don't ask, tell principle defines that in your object, you should tell another object what to do, and not ask about the other object's state and then do the work by yourself in your object.*

If your object asks many things from another object using, e.g., multiple getters, you might be guilty of the *feature envy* design smell. Your object is envious of a feature that the other object should

have.

Let's have an example and define a cube shape class:

```
public interface ThreeDShape {
  // ...
}

public class Cube3DShape implements ThreeDShape {
  private final int width;
  private final int height;
  private final int depth;

  // Constructor...

  public int getWidth() {
    return width;
  }

  public int getHeight() {
    return height;
  }

  public int getDepth() {
    return depth;
  }
}
```

Next, we define another class, `CubeUtils`, that contains a method for calculating the total volume of cubes:

```
public class CubeUtils {
  public int calculateTotalVolume(
    final List<Cube3DShape> cubes
  ) {
    int totalVolume = 0;

    for (final Cube3DShape cube : cubes) {
      final var width = cube.getWidth();
      final var height = cube.getHeight();
      final var depth = cube.getDepth();
      totalVolume += width * height * depth;
    }

    return totalVolume;
  }
}
```

In the `calculateTotalVolume` method, we ask three times about a cube object's state. This is against the *don't ask, tell principle*. Our method is envious of the volume calculation feature and wants to do it by itself rather than telling a `Cube3DShape` object to calculate its volume.

Let's correct the above code so that it follows the *don't ask, tell principle*:

```
public interface ThreeDShape {
  int calculateVolume();
}
```

```
public class Cube3DShape implements ThreeDShape {
  private final int width;
  private final int height;
  private final int depth;

  // Constructor

  public int calculateVolume() {
    return height * width * depth;
  }
}

public class ThreeDShapeUtils {
  public int calculateTotalVolume(
    final List<ThreeDShape> threeDShapes
  ) {
    int totalVolume = 0;

    for (final var threeDShape : threeDShapes) {
      totalVolume += threeDShape.calculateVolume();
    }

    return totalVolume;
  }
}
```

Now our `calculateTotalVolume` method is not asking anything about a cube object. It just tells a cube object to calculate its volume. We also removed the *asking* methods (getters) from the `Cube3DShape` class because they are no longer needed.

Below is another example of asking instead of telling:

```
using namespace std::chrono_literals;
using std::chrono::system_clock;

void AnomalyDetectionEngine::runEngine()
{
  while (m_isRunning)
  {
    const auto now = system_clock::now();

    if (m_anomalyDetector->shouldDetectAnomalies(now))
    {
      const auto anomalies = m_anomalyDetector->detectAnomalies();
      // Do something with the detected anomalies
    }

    std::this_thread::sleep_for(1s);
  }
}
```

In the above example, we ask the anomaly detector if we should detect anomalies now. Then, depending on the result, we call another method on the anomaly detector to detect anomalies. This could be simplified by making the `detectAnomalies` method to check if anomalies should be detected using the `shouldDetectAnomalies` method. Then the `shouldDetectAnomalies` method can be made private, and we can simplify the above code as follows:

```
using namespace std::chrono_literals;

void AnomalyDetectionEngine::runEngine()
{
  while (m_isRunning)
  {
    const auto anomalies = m_anomalyDetector->detectAnomalies();
    // Do something with the detected anomalies
    std::this_thread::sleep_for(1s);
  }
}
```

Law of Demeter

A method on an object received from another object's method call should not be called.

The below statements are considered to break the law:

```
user.getAccount().getBalance();
user.getAccount().withdraw(...);
```

The above statements can be corrected either by moving functionality to a different class or by making the second object to act as a facade between the first and the third object.

Below is an example of the latter solution, where we introduce two new methods in the `User` class and remove the `getAccount` method:

```
user.getAccountBalance();
user.withdrawFromAccount(...);
```

In the above example, the `User` class is a facade in front of the `Account` class that we should not access directly from our object. However, you should always check if the first solution alternative could be used instead. It makes the code more object-oriented and does not require creating additional methods.

Below is a Java example that uses `User` and `SalesItem` entities and is not obeying the law of Demeter:

```
void purchase(final User user, final SalesItem salesItem) {
  final var account = user.getAccount();

  // Breaks the law
  final var accountBalance = account.getBalance();

  final var salesItemPrice = salesItem.getPrice();

  if (accountBalance >= salesItemPrice) {
    account.withdraw(salesItemPrice); // Breaks the law
  }
```

```
  // ...
}
```

We can resolve the problem in the above example by moving the `purchase` method to the correct class, in this case, the `User` class:

```
class User {
  private Account account;

  // ...

  void purchase(final SalesItem salesItem) {
    final var accountBalance = account.getBalance();
    final var salesItemPrice = salesItem.getPrice();

    if (accountBalance >= salesItemPrice) {
      account.withdraw(salesItemPrice);
    }

    // ...
  }
}
```

Avoid Primitive Type Obsession Principle

> *Avoid primitive type obsession by defining semantic types for function parameters and function return value.*

Many of us have experienced situations where we have supplied arguments to a function in the wrong order. This is easy if the function, for example, takes two integer parameters, but you accidentally give those two integer parameters in the wrong order. You don't get a compilation error.

Another problem with primitive types as function arguments is that the argument values are not necessarily validated. You have to implement the validation logic in your function.

Suppose you accept an integer parameter for a port number in a function. In that case, you might get any integer value as the parameter value, even though the valid port numbers are from 1 to 65535. Suppose you also had other functions in the same codebase accepting a port number as a parameter. In that case, you could end up doing the same validation logic in multiple places and have thus duplicate code in your codebase.

Let's have a simple example of using this principle:

RectangleShape.java

```
public class RectangleShape implements Shape {
  private int width;
  private int height;
```

```
  public RectangleShape(final int width, final int height) {
    this.width = width;
    this.height = height;
  }
}
```

In the above example, the constructor has two parameters with the same primitive type (int). It is possible to give `width` and `height` in the wrong order. But if we refactor the code to use objects instead of primitive values, we can make the likelihood of giving the arguments in the wrong order much smaller:

```
public class Value<T> {
  private final T value;

  public Value(final T value) {
    this.value = checkNotNull(value);
  }

  T get() {
    return value;
  }
}

public class Width extends Value<Integer> {
  public Width(final int width) {
    super(width);
  }
}

public class Height extends Value<Integer> {
  public Height(final int height) {
    super(height);
  }
}

public class RectangleShape implements Shape {
  private final int width;
  private final int height;

  public RectangleShape(final Width width, final Height height) {
    this.width = width.get();
    this.height = height.get();
  }
}

final var width = new Width(20);
final var height = new Height(50);

// OK
final Shape rectangle = new RectangleShape(width, height);

// Does not compile, parameters are in wrong order
final Shape rectangle2 = new RectangleShape(height, width);

// Does not compile, first parameter is not a width
final Shape rectangle3 = new RectangleShape(height, height);

// Does not compile, second parameter is not a height
final Shape rectangle4 = new RectangleShape(width, width);
```

```
// Does not compile, Width and Height objects must be used
// instead of primitive types
final Shape rectangle5 = new RectangleShape(20, 50);
```

In the above example, `Width` and `Height` are simple data classes. They don't contain any behavior. You can use concrete data classes as function parameter types. There is no need to create an interface for a data class. So, the *program against interfaces* principle does not apply here.

Let's have another simple example where we have the following function signature:

```
public void doSomething(final String namespacedName, ...) {
  // ...
}
```

The above function signature allows function callers to supply a non-namespaced name accidentally. By using a custom type for the namespaced name, we can formulate the above function signature to the following:

```
public class NamespacedName {
  private final String namespacedName;

  public NamespacedName(
    final String namespace,
    final String name
  ) {
    this.namespacedName = namespace.isEmpty()
                          ? name
                          : (namespace + '.' + name);
  }

  public String get() {
    return this.namespacedName;
  }
}

public void doSomething(final NamespacedName namespacedName, ...) {
  // ...
}
```

Let's have a more comprehensive example with an `HttpUrl` class. The class constructor has several parameters that should be validated upon creating an HTTP URL:

HttpUrl.java

```
public class HttpUrl {
  private final String httpUrl;

  public HttpUrl(
    final String scheme,
    final String host,
    final int port,
    final String path,
    final String query
  ) {
    httpUrl = scheme +
```

```
              "://" +
              host +
              ":" +
              port +
              path +
              "?" +
              query;
  }
}
```

Let's introduce an abstract class for validated values:

AbstractValidatedValue.java

```
public abstract class AbstractValidatedValue<T> {
  protected final T value;

  public AbstractValidatableValue(final T value) {
    this.value = checkNotNull(value);
  }

  abstract boolean valueIsValid();

  Optional<T> get() {
    return valueIsValid()
              ? Optional.of(value)
              : Optional.empty();
  }

  T tryGet() {
    if (valueIsValid()) {
      return value;
    } else {
      throw new ValidatedValueGetError(...);
    }
  }
}
```

Let's create a class for validated HTTP scheme objects:

HttpScheme.java

```
public class HttpScheme extends AbstractValidatedValue<String> {
  public HttpScheme(String value) {
    super(value);
  }

  public boolean valueIsValid() {
    // Because the AbstractValidatedValue<String> is immutable,
    // if you had complex validation logic, you could cache
    // the validation result and store it to a class attribute.

    return "https".equals(value.toLowerCase()) ||
           "http".equals(value.toLowerCase());
  }
}
```

Let's create a `Port` class (and similar classes for the host, path, and query should be created):

```
public class Port extends AbstractValidatedValue<Integer> {
  public Port(Integer value) {
    super(value);
  }

  public boolean valueIsValid() {
    return value >= 1 && value <= 65535;
  }
}

// public class Host ...
// public class Path ...
// public class Query ...
```

Let's create a utility class, `OptionalUtils`, with a method for mapping a result for five optional values:

```
@FunctionalInterface
public interface Mapper<T, U, V, X, Y, R> {
  R map(T value,
        U value2,
        V value3,
        X value4,
        Y value5);
}

public final class OptionalUtils {
  public static <T, U, V, X, Y, R> Optional<R>
    mapAll(
      final Optional<T> opt1,
      final Optional<U> opt2,
      final Optional<V> opt3,
      final Optional<X> opt4,
      final Optional<Y> opt5,
      final Mapper<T, U, V, X, Y, R> mapper
  ) {
    if (opt1.isPresent() &&
        opt2.isPresent() &&
        opt3.isPresent() &&
        opt4.isPresent() &&
        opt5.isPresent()
    ) {
      return Optional.of(mapper.map(opt1.get(),
                                    opt2.get(),
                                    opt3.get(),
                                    opt4.get(),
                                    opt5.get()));
    } else {
      return Optional.empty();
    }
  }
}
```

Next, we can reimplement the `HttpUrl` class to contain two alternative factory methods for creating an HTTP URL:

HttpUrl.java

```
public class HttpUrl {
```

```
  private final String httpUrl;

  // Constructor is private because factory methods
  // should be used to create instances of this class
  private HttpUrl(final String httpUrl) {
    this.httpUrl = httpUrl;
  }

  // Factory method that returns an optional HttpUrl
  public static Optional<HttpUrl> create(
    final HttpScheme scheme,
    final Host host,
    final Port port,
    final Path path,
    final Query query
  ) {
    return OptionalUtils.mapAll(scheme.get(),
                                host.get(),
                                port.get(),
                                path.get(),
                                query.get(),
            (schemeValue,
             hostValue,
             portValue,
             pathValue,
             queryValue) ->
                    new HttpUrl(schemeValue +
                                "://" +
                                hostValue +
                                ":" +
                                portValue +
                                pathValue +
                                "?" +
                                queryValue));
  }

  // Factory method that returns a valid HttpUrl or
  // throws an error
  public static HttpUrl tryCreate(
    final HttpScheme scheme,
    final Host host,
    final Port port,
    final Path path,
    final Query query
  ) {
    try {
      return new HttpUrl(scheme.tryGet() +
                         "://" +
                         host.tryGet() +
                         ":" +
                         port.tryGet() +
                         path.tryGet() +
                         "?" +
                         query.tryGet());
    } catch (final ValidatedValueGetError error) {
      throw new HttpUrlCreateError(error);
    }
  }
}
```

Notice how we did not hardcode the URL validation inside the `HttpUrl` class, but we created small

validated value classes: `HttpScheme`, `Host`, `Port`, `Path`, and `Query`. These classes can be further utilized in other parts of the codebase if needed and can even be put into a common validation library for broader usage.

For TypeScript, I have created a library called *validated-types* for easily creating and using semantically validated types. The library is available at *https://github.com/pksilen/validated-types*. The library's idea is to validate data when the data is received from the input. You can then pass already validated, strongly typed data to the rest of the functions in your software component.

An application typically receives unvalidated input data from external sources in the following ways:

- Reading command line arguments
- Reading environment variables
- Reading standard input
- Reading files from the file system
- Reading data from a socket (network input)
- Receiving input from a user interface (UI)

Below is an example of using the *validated-types* library to create a validated integer type that allows values between 1 and 10. The `VInt` generic type takes a type argument of string type, which defines the allowed value range in the following format: *<min-value>,<max-value>*

```
import { VInt } from 'validated-types';

function useInt(int: VInt<'1,10'>) {
  // The wrapped integer value can be accessed
  // through the 'value' property
  console.log(int.value);
}

const int: VInt<'1,10'> = VInt.tryCreate('1,10', 5);
useInt(int); // prints to console: 5

// Returns null, because 12 is not between 1 and 10
const maybeInt: VInt<'1,10'> | null = VInt.create('1,10', 12);

// Prints to console: 10
useInt(maybeInt ?? VInt.tryCreate('1,10', 10));

// Throws, because 500 is not between 1 and 10
const int2: VInt<'1,10'> = VInt.tryCreate('1,10', 500);
```

The below example defines a `Url` type which contains six validations that validate a string matching the following criteria:

- is at least one character long
- is at most 1024 characters long

- is a lowercase string
- is a valid URL
- URL starts with 'https'
- URL ends with '.html'

```
import { SpecOf, VString } from 'validated-types';

// First element in the VString type parameter array validates
// a lowercase string between 1-1024 characters long

// Second element in the VString type parameter array validates
// an URL

// Third element in the VString type parameter array validates
// a string that starts with "https"

// Fourth element in the VString type parameter array validates
// a string that ends with ".html"

type Url = VString<['1,1024,lowercase',
                    'url',
                    'startsWith,https',
                    'endsWith,.html']>;

const urlVSpec: VSpecOf<Url> = ['1,1024,lowercase',
                                'url',
                                'startsWith,https',
                                'endsWith,.html'];

function useUrl(url: Url) {
  console.log(url.value);
}

const url: Url = VString.tryCreate(
  urlVSpec,
  'https://server.domain.com:8080/index.html'
);

// Prints to console: https://server.domain.com:8080/index.html
useUrl(url);

// 'maybeUrl' will be null
const maybeUrl: Url | null = VString.create(urlVSpec,
                                            'invalid URL');

const defaultUrl: Url = VString.tryCreate(
  urlVSpec,
  'https://default.domain.com:8080/index.html'
);

// Prints to console: https://default.domain.com:8080/index.html
useUrl(maybeUrl ?? defaultUrl);
```

If you don't need validation but would like to create a semantic type, you can use the `SemType` class from the *validated-types* library:

```
import { SemType } from 'validated-types';
```

```
// Defines a semantic boolean type with name 'isRecursiveCall'
type IsRecursiveCall = SemType<boolean, 'isRecursiveCall'>

// Defines a semantic boolean type with name 'isInternalCall'
type IsInternalCall = SemType<boolean, 'isInternalCall'>;

function myFunc(isRecursiveCall: IsRecursiveCall,
                isInternalCall: IsInternalCall) {
  // The value of a semantic type variable
  // can be obtained from the 'value' property
  console.log(isRecursiveCall.value);
  console.log(isInternalCall.value);
}

const isRecursiveCall = false;
const isInternalCall = true;

// This will succeed
myFunc(new SemType({ isRecursiveCall }),
       new SemType({ isInternalCall }));

// All the below myFunc calls will fail during
// the compilation
myFunc(new SemType({ isInternalCall }),
       new SemType({ isRecursiveCall }));

myFunc(true, true);

myFunc(new SemType('isSomethingElse', true),
       new SemType('isInternalCall', true));

myFunc(new SemType('isRecursiveCall', false),
       new SemType('isSomethingElse', true));

myFunc(new SemType('isSomethingElse', true),
       new SemType('isSomethingElse', true));
```

Dependency Injection (DI) Principle

> *Dependency injection (DI) allows changing the behavior of an application based on static or dynamic configuration. When using dependency injection, the dependencies are injected only upon the application startup. The application can first read its configuration and then decide what objects are created for the application. In many languages, dependency injection is crucial for unit tests also. When executing a unit test using DI, you can inject mock dependencies into the tested code instead of using the standard dependencies of the application.*

Below is a C++ example of using the singleton pattern without dependency injection:

main.cpp

```
int main()
{
  Logger::initialize("Exporter");
```

```
  Logger::writeLogEntry(LogLevel::Info,
                        std::source_location::current(),
                        "Starting application");

  // ...
}
```

A developer must remember to call the `initialize` method before calling any other method on the `Logger` class. This kind of coupling between methods should be avoided. Also, it is hard to unit test the static methods of the `Logger` class.

We should refactor the above code to use dependency injection:

main.cpp

```
int main()
{
  DependencyInjectorFactory::createDependencyInjector(...)
    ->injectDependencies();

  Logger::getInstance()->writeLogEntry(
    LogLevel::Info,
    std::source_location::current(),
    "Starting application"
  );

  // ...
}
```

Singleton.h

```
template<typename T>
class Singleton
{
public:
  Singleton() = default;

  virtual ~Singleton()
  {
    m_instance.reset();
  };

  static inline std::shared_ptr<T>& getInstance()
  {
    return m_instance;
  }

  static void setInstance(const std::shared_ptr<T>& instance)
  {
    m_instance = instance;
  }

private:
  static inline std::shared_ptr<T> m_instance;
};
```

Logger.h

```
class Logger : public Singleton<Logger>
{
public:
  virtual void writeLogEntry(...) = 0;
};
```

StdOutSyslogLogger.h

```
class StdOutSyslogLogger : public Logger
{
public:
  void writeLogEntry(...) override
  {
    // Write the log entry
    // in Syslog format to the standard output
  }
};
```

DependencyInjectorFactory.h

```
class DependencyInjectorFactory {
public:
  static std::shared_ptr<DependencyInjector>
  createDependencyInjector(...)
  {
    // You can use a switch-case here to create
    // different kinds of dependency injectors
    // that inject different kinds of dependencies
    return std::make_shared<DefaultDependencyInjector>();
  }
}
```

DependencyInjector.h

```
class DependencyInjector {
public:
  virtual ~DependencyInjector = default;
  virtual void injectDependencies() = 0;
}
```

DefaultDependencyInjector.h

```
class DefaultDependencyInjector : public DependencyInjector {
public:
  void injectDependencies() override;
}
```

DefaultDependencyInjector.cpp

```
void DefaultDependencyInjector::injectDependencies()
{
  // Inject other dependencies...

  Logger::setInstance(
    std::make_shared<StdOutSyslogLogger>("Exporter")
  );
}
```

Below is an example of a *data-visualization-web-client* where the *noicejs* NPM library is used for

dependency injection. This library is similar to the Google Guice library. Below is a `FakeServicesModule` class that configures dependencies for different backend services that the web client uses. As you can notice, all the services are configured to use fake implementations because this DI module is used when the backend services are not yet available. A `RealServicesModule` class can be implemented and used when the backend services become available. In the `RealServicesModule` class, the services are bound to their actual implementation classes instead of fake implementations.

FakeServicesModule.ts

```
import { Module } from 'noicejs';
import FakeDataSourceService from ...;
import FakeMeasureService from ...;
import FakeDimensionService from ...;
import FakeChartDataService from ...;

export default class FakeServicesModule extends Module {
  override async configure(): Promise<void> {
    this.bind('dataSourceService')
      .toInstance(new FakeDataSourceService());

    this.bind('measureService')
      .toInstance(new FakeMeasureService());

    this.bind('dimensionService')
      .toInstance(new FakeDimensionService());

    this.bind('chartDataService')
      .toInstance(new FakeChartDataService());
    );
  }
}
```

With the *noicejs* library, you can configure several DI modules and create a DI container from the wanted modules. The module approach lets you divide dependencies into multiple modules, so you don't have a single big module. It also lets you instantiate a different module or modules based on the application configuration.

In the below example, the DI container is created from a single module, an instance of the `FakeServicesModule` class:

diContainer.ts

```
import { Container } from 'noicejs';
import FakeServicesModule from './FakeServicesModule';

const diContainer = Container.from(new FakeServicesModule());

export default diContainer;
```

In the development phase, we could create two separate modules, one for fake services and another one for real services, and control the application behavior based on the web page's URL query parameter:

diContainer.ts

```
import { Container } from 'noicejs';
import FakeServicesModule from './FakeServicesModule';
import RealServicesModule from './RealServicesModule';

const diContainer = (() => {
  if (location.href.includes('useFakeServices=true')) {
    // Use fake services if web page URL
    // contains 'useFakeServices=true'
    return Container.from(new FakeServiceModule());
  } else {
    // Otherwise use real services
    return Container.from(new RealServicesModule());
  }
})();

export default diContainer;
```

Then you must configure the `diContainer` before dependency injection can be used. In the below example, the `diContainer` is configured before a React application is rendered:

index.tsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import diContainer from './diContainer';
import AppView from './app/view/AppView';

diContainer.configure().then(() => {
  ReactDOM.render(<AppView />, document.getElementById('root'));
});
```

Then, in Redux actions, where you need a service, you can inject the required service with the `@Inject` decorator. You specify the name of the service you want to inject. The service will be injected as the class constructor argument's property (with the same name).

StartFetchChartDataAction.ts

```
// Imports ...

type ConstructorArgs = {
  chartDataService: ChartDataService,
  chart: Chart,
  dispatch: Dispatch;
};

export default
@Inject('chartDataService')
class StartFetchChartDataAction extends AbstractChartAreaAction {
  private readonly chartDataService: ChartDataService;
  private readonly chart: Chart;

  constructor({ chart,
                chartDataService,
                dispatch }: ConstructorArgs) {
    super(dispatch);
    this.chartDataService = chartDataService;
    this.chart = chart;
```

```
  }

  perform(currentState: ChartAreaState): ChartAreaState {
    this.chartDataService
      .fetchChartData(
        this.chart.dataSource,
        this.chart.getColumns(),
        this.chart.getSelectedFilters(),
        this.chart.getSelectedSortBys()
      )
      .then((columnNameToValuesMap: ColumnNameToValuesMap) => {
        this.dispatch(
          new FinishFetchChartDataAction(columnNameToValuesMap,
                                         this.chart.id)
        );
      })
      .catch((error) => {
        // Handle error
      });

    this.chart.isFetchingChartData = true;
    return ChartAreaStateUpdater
            .getNewStateForChangedChart(currentState, this.chart);
  }
}
```

And to be able to dispatch the above action, a controller should be implemented:

ChartAreaController.ts

```
import diContainer from './diContainer';
import StartFetchChartDataAction from './StartFetchChartDataAction';
import Controller, { Dispatch } from './Controller';

class ChartAreaController extends Controller {
  constructor(dispatch: Dispatch) {
    super(dispatch)
  }

  getActionDispatchers() {
    return {
      startFetchChartData: (chart: Chart) =>
        // the 'chart' is given as a property to
        // StartFetchChartDataAction class constructor
        this.dispatchWithDi(diContainer,
                            StartFetchChartDataAction,
                            { chart });
    }
  }
}
```

The following base classes are also defined:

AbstractAction.ts

```
export default abstract class AbstractAction<S> {
  abstract perform(state: S): S;
}
```

AbstractDispatchingAction.ts

```
// Imports...

export default abstract class AbstractDispatchingAction<S>
    extends AbstractAction<S> {
  constructor(protected readonly dispatch: Dispatch) {}
}
```

AbstractChartAreaAction.ts

```
// Imports...

export default abstract class AbstractChartAreaAction
         extends AbstractDispatchingAction<ChartAreaState> {
}
```

Controller.ts

```
export type Dispatch =
  (plainActionObject: { type: AbstractAction<any> }) => void;

export default class Controller {
  protected readonly dispatch:
    (action: AbstractAction<any>) => void;

  // The 'dispatch' is from Redux library
  constructor(dispatch: Dispatch) {
    this.dispatch = (action: AbstractAction<any>) =>
      dispatch({ type: action });
  }

  dispatchWithDi(
    diContainer: { create: (...args: any[]) => Promise<any> },
    ActionClass:
      abstract new (...args: any[]) => AbstractAction<any>,
    otherArgs: {}
  ) {
    // diContainer.create will create a new object of
    // class ActionClass.
    // The second parameter of the create function defines
    // additional properties supplied to ActionClass constructor.
    // The create method is asynchronous. When it succeeds,
    // the created action object is available in the 'then'
    // function and it can be now dispatched

    diContainer
      .create(ActionClass, {
        dispatch: this.dispatch,
        ...otherArgs
      })
      .then((action: any) => this.dispatch(action));
  }
}
```

Avoid Code Duplication Principle

At the class level, when you spot duplicated code in two different classes implementing the same interface, you should create a new base class to accommodate the common

functionality and make the classes extend the new base class.

Below is an `AvroBinaryKafkaInputMessage` class that implements the `InputMessage` interface:

InputMessage.h

```
class InputMessage
{
public:
  virtual ~InputMessage() = default;

  virtual uint32_t tryDecodeSchemaId() const = 0;

  virtual std::shared_ptr<DecodedMessage>
  tryDecodeMessage(const std::shared_ptr<Schema>& schema)
  const = 0;
};
```

AvroBinaryKafkaInputMessage.h

```
class AvroBinaryKafkaInputMessage : public InputMessage
{
public:
  AvroBinaryKafkaInputMessage(
    std::unique_ptr<RdKafka::Message> kafkaMessage
  ) : m_kafkaMessage(std::move(kafkaMessage))
  {}

  uint32_t tryDecodeSchemaId() const override;

  std::shared_ptr<DecodedMessage>
  tryDecodeMessage(const std::shared_ptr<Schema>& schema)
  const override;

private:
  std::unique_ptr<RdKafka::Message> m_kafkaMessage;
};

uint32_t AvroBinaryKafkaInputMessage::tryDecodeSchemaId() const
{
  // Try decode schema id from the beginning of
  // the Avro binary Kafka message
}

std::shared_ptr<DecodedMessage>
AvroBinaryKafkaInputMessage::tryDecodeMessage(
  const std::shared_ptr<Schema>& schema
) const
{
  return schema->tryDecodeMessage(m_kafkaMessage->payload(),
                                  m_kafkaMessage->len());
}
```

If we wanted to introduce a new Kafka input message class for JSON, CSV, or XML format, we could create a class like the `AvroBinaryKafkaInputMessage` class. But then we can notice the duplication of code in the `tryDecodeMessage` method. We can notice that the `tryDecodeMessage` method is the same regardless of the input message source and format.

According to this principle, we should move the duplicate code to a common base class, `BaseInputMessage`. We could make the `tryDecodeMessage` method a template method according to the *template method pattern* and create abstract methods for getting the message data and its length:

BaseInputMessage.h

```
class BaseInputMessage : public InputMessage
{
public:
  std::shared_ptr<DecodedMessage>
  tryDecodeMessage(const std::shared_ptr<Schema>& schema)
  const final;

protected:
  // Abstract methods
  virtual uint8_t* getData() const = 0;
  virtual size_t getLengthInBytes() const = 0;
};

// This is a template method
// 'getData' and 'getLengthInBytes' will be
// implemented in subclasses
std::shared_ptr<DecodedMessage>
BaseInputMessage::tryDecodeMessage(
  const std::shared_ptr<Schema>& schema
) const
{
  return schema->tryDecodeMessage(getData(), getLengthInBytes());
}
```

Next, we should refactor the `AvroBinaryKafkaInputMessage` class to extend the new `BaseInputMessage` class and implement the `getData` and `getLengthInBytes` methods. But we can realize these two methods are the same for all Kafka input message data formats. We should not implement those two methods in the `AvroBinaryKafkaInputMessage` class because we would need to implement them as duplicates if we needed to add a Kafka input message class for another data format. Once again, we can utilize this principle and create a new base class for Kafka input messages:

KafkaInputMessage.h

```
class KafkaInputMessage : public BaseInputMessage
{
public:
  KafkaInputMessage(
    std::unique_ptr<RdKafka::Message> kafkaMessage
  ) : m_kafkaMessage(std::move(kafkaMessage))
  {}

protected:
  uint8_t* getData() const final;
  size_t getLengthInBytes() const final;

private:
  std::unique_ptr<RdKafka::Message> m_kafkaMessage;
};
```

```
uint8_t* KafkaInputMessage::getData() const
{
  return std::bit_cast<uint8_t*>(m_kafkaMessage->payload());
}

size_t KafkaInputMessage::getLengthInBytes() const
{
  return m_kafkaMessage->len();
}
```

Finally, we can refactor the `AvroBinaryKafkaInputMessage` class to contain no duplicated code:

AvroBinaryKafkaInputMessage.h

```
class AvroBinaryKafkaInputMessage : public KafkaInputMessage
{
public:
  uint32_t tryDecodeSchemaId() const final;
};

uint32_t AvroBinaryKafkaInputMessage::tryDecodeSchemaId() const
{
  // Try decode the schema id from the beginning of
  // the Avro binary Kafka message
  // Use base class getData() and getDataLengthInBytes()
  // methods to achieve that
}
```

Inheritance in Cascading Style Sheets (CSS)

In HTML, you can define classes (class names) for HTML elements:

```
<span class="icon pie-chart-icon">...</span>
```

In a CSS file, you define CSS properties for CSS classes, for example:

```
.icon {
  background-repeat: no-repeat;
  background-size: 1.9rem 1.9rem;
  display: inline-block;
  height: 2rem;
  margin-bottom: 0.2rem;
  margin-right: 0.2rem;
  width: 2rem;
}

.pie-chart-icon {
  background-image: url('pie_chart_icon.svg');
}
```

The problem with the above approach is that it is not correctly object-oriented. In the HTML code, you must list all the class names to achieve a mixin of all the needed CSS properties. It is easy to

forget to add a class name. For example, you could specify `pie-chart-icon` only and forget to specify the `icon`.

It is also difficult to change the inheritance hierarchy afterward. Suppose you wanted to add a new class `chart-icon` for all the chart icons:

```
.chart-icon {
  // Define properties here...
}
```

You would have to remember to add the `chart-icon` class name to all places in the HTML code where you are rendering chart icons:

```
<span class="icon chart-icon pie-chart-icon">...</span>
```

The above-described approach is very error-prone. What you should do is introduce proper object-oriented design. You need a CSS preprocessor that makes extending CSS classes possible. In the below example, I am using SCSS:

```
<span class="pieChartIcon">...</span>
```

```
.icon {
  background-repeat: no-repeat;
  background-size: 1.9rem 1.9rem;
  display: inline-block;
  height: 2rem;
  margin-bottom: 0.2rem;
  margin-right: 0.2rem;
  width: 2rem;
}

.chartIcon {
  @extend .icon;

  // Other chart icon related properties...
}

.pieChartIcon {
  @extend .chartIcon;

  background-image: url('../../../../../assets/images/icons/chart/pie_chart_icon.svg');
}
```

In the above example, we define only one class for the HTML element. The inheritance hierarchy is defined in the SCSS file using the `@extend` directive. We are now free to change the inheritance hierarchy in the future without any modification needed in the HTML code.

Coding Principles

This chapter presents principles for coding. The following principles are presented:

- Uniform variable naming principle
- Uniform source code repository structure principle
- Source code directory tree structure principle
- Avoid comments principle
- Function single return statement principle
- Prefer statically typed language principle
- Refactoring principle
- Static code analysis principle
- Error/Exception handling principle
- Don't pass or return null principle
- Avoid off-by-one errors principle
- Be critical when googling principle
- Optimization principle

Uniform Variable Naming Principle

> *A good variable name should describe the variable's purpose and its type.*

At best, having your code written with great names makes it read like prose. And remember that code is more often read than written, so code must be easy to read and understand.

Naming variables with names that also convey information about the variable's type is crucial in untyped languages and beneficial in typed languages, too, because modern typed languages use automatic type deduction, and you won't always see the actual type of a variable. But when the

variable's name tells its type, it does not matter if the type name is not visible.

In the following sections, naming conventions for different types of variables are proposed.

Naming Integer Variables

Some variables are intrinsically integers, like *age* or *year*. Everybody understands immediately that the type of an *age* or *year* variable is a number and, to be more specific, an integer. So you don't have to add anything to the variable's name to indicate its type. It already tells you its type.

One of the most used categories of integer variables is a count or number of something. You see those kinds of variables in every piece of code. I recommend using the following convention for naming those variables: *numberOf<something>* or alternatively *<something>Count*. For example, *numberOfFailures* or *failureCount*. You should not use a variable name *failures* to designate a failure count. The problem with that variable name is it does not clearly specify the type of the variable and thus can cause some confusion. This is because a variable named *failures* can be misunderstood as a collection variable (e.g., a list of failures).

If the unit of a variable is not self-evident, always add information about the unit to the end of the variable name. For example, instead of naming a variable *tooltipShowDelay*, you should name it *tooltipShowDelayInMillis* or *tooltipShowDelayInMillisecs*. If you have a variable whose unit is self-evident, unit information is not needed. So, there is no need to name an *age* variable as *ageInYears*. But if you are measuring age in months, you must name the respective variable as *ageInMonths* so that people don't assume that age is measured in years.

Naming Floating-Point Number Variables

Floating-point numbers are not as common as integers, but sometimes you need them too. Some values are intrinsically floating-point numbers, like most un-rounded measures (e.g., price, height, width, or weight). If you need to store a measured value, it would be a safe bet to have a floating-point variable.

If you need to store an amount of something that is not an integer, use a variable named *<something>Amount*, like *rainfallAmount*. When you see "amount of something" in code, you can automatically think it is a floating-point number. If you need to use a number in arithmetic, depending on the application, you might want to use either floating-point or integer arithmetic. In the case of money, you should use integer arithmetic to avoid rounding errors. Instead of a floating-point *moneyAmount* variable, you should have an integer variable, like *moneyInCents*, for example.

If the unit of a variable is not self-evident, add information about the unit to the end of the variable name, like *rainfallAmountInMillimeters*, *widthInInches*, *angleInDegrees* (values 0-360),

failurePercent (values 0-100), or *failureRatio* (values 0-1).

Naming Boolean Variables

Boolean variables can have only one of two values: true or false. The name of a boolean variable should form a statement where the answer is true or false, or yes or no. Typical boolean variable naming patterns are: *is<something>*, *has<something>*, *did<something>*, *should<something>*, *can<something>*, or *will<something>*. Some examples of variable names following the above patterns are *isDisabled*, *hasErrors*, *didUpdate*, *shouldUpdate*, and *willUpdate*.

The verb in the boolean variable name does not have to be at the beginning. It can and should be in the middle if it makes the code read better. Boolean variables are often used in if-statements where changing the word order in the variable name can make the code read better. Remember that, at best, code reads like beautiful prose, and code is read more often than written.

Below is a code snippet where we have a boolean variable named `isPoolFull`:

```
if (const bool isPoolFull = m_pooledMessages.size() >= 200U;
      isPoolFull)
{
  // ...
}
else
{
  // ...
}
```

We can change the variable name to `poolIsFull` to make the if-statement read more fluently. In the below example, the if-statements reads "if pollIsFull" instead of "if isPoolFull”:

```
if (const bool poolIsFull = m_pooledMessages.size() >= 200U;
      poolIsFull)
{
  // ...
```

Don't use boolean variable names in the form of *<passive-verb><Something>*, like *insertedField*, because this can confuse the reader. It is unclear if the variable name is a noun that names an object or a boolean statement. Instead, use either *didInsertField* or *fieldWasInserted*.

Below is a Go language example of the incorrect naming of a variable used to store a function return value. Someone might think that `tablesDropped` means a list of dropped table names. So, the name of the variable is obscure and should be changed.

```
tablesDropped := dropRedundantTables(prefix,
                                     vmsdata,
                                     cfg.HiveDatabase,
                                     hiveClient,
                                     logger)
if tablesDropped {
  // ...
}
```

Below is the above example modified so that the variable name is changed to indicate a boolean statement:

```
tablesWereDropped := dropRedundantTables(prefix,
                                         vmsdata,
                                         cfg.HiveDatabase,
                                         hiveClient,
                                         logger)
if tablesWereDropped {
  // ...
}
```

You could have used a variable named `didDropTables`, but the `tablesWereDropped` makes the if-statement more readable.

Naming String Variables

String variables are prevalent, and many things are intrinsically strings, like *name*, *title*, *city*, *country*, or *topic*. When you need to store numerical data in a string variable, tell the code reader clearly that it is a question about a number in string format, and use a variable name in the following format: *<someValue>String* or *<someValue>AsString*. It makes the code more prominent and easier to understand. For example:

```
const year = parseInt(yearAsString, 10);
```

Naming Enum Variables

Name enum variables with the same name as the enum type. E.g., a `CarType` enum variable should be named `carType`. If the name of an enum type is very generic, like `Result`, you might benefit from declaring an enum variable with some detail added to the variable name. Below is an example of a very generic enum type name:

PulsarProducer.cpp

```
// Returns enum type 'Result'
const auto result = pulsar::createProducer(...);

if (result == Result.Ok) {
  // ...
}
```

Let's add some detail and context to the `result` variable name:

PulsarProducer.cpp

```
const auto producerCreationResult = pulsar::createProducer(...);

if (producerCreationResult == Result.Ok) {
  // ...
}
```

Naming Collection (Array, List, and Set) Variables

When naming arrays, lists, and sets, you should use the plural form of a noun, like *customers*, *errors*, or *tasks*. In most cases, this is enough because you don't necessarily need to know the underlying collection implementation. Using this naming convention allows you to change the type of a collection variable without needing to change the variable name. If you are iterating over a collection, it does not matter if it is an array, list, or set. Thus, it does not bring any benefit if you add the collection type name to the variable name, for example, *customerList* or *taskSet*. Those names are just longer. You might want to specify the collection type in some special cases. Then, you can use the following kind of variable names: *queueOfTasks*, *stackOfCards*, or *orderedSetOfTimestamps*.

Below is an example in Go language, where the function is named correctly to return a collection (of categories), but the variable receiving the return value is not named according to the collection variable naming convention:

```
vmsdata, error = vmsClient.GetCategories(vmsUrl, logger)
```

Correct naming would be:

```
vmsCategories, error = vmsClient.GetCategories(vmsUrl, logger)
```

Naming Map Variables

Maps are accessed by requesting a *value* for a certain *key*. This is why I recommend naming maps using the pattern *<key>To<Value>Map*. Let's say we have a map containing order counts for customer ids. This map should be named `customerIdToOrderCountMap`. Or if we have a list of suppliers for product names, the map variable should be named `productNameToSuppliersMap`. Below is an example of accessing maps in Java:

```
final var orderCount = customerIdToOrderCountMap.get(customerId);
final var suppliers = productNameToSuppliersMap.get(productName);
```

Below is an example of iterating over a map in JavaScript:

```
Object.entries(customerIdToOrderCountMap)
      .map(([customerId, orderCount]) => ...);
```

Naming Pair and Tuple Variables

A variable containing a pair should be named using the pattern *variable1AndVariable2*. For example: `heightAndWidth`. And for tuples, the recommended naming pattern is *variable1Variable2...andVariableN*. For instance: `heightWidthAndDepth`.

Below is an example of using pairs and tuples in JavaScript:

```
const heightAndWidth = [100, 200];
const heightWidthAndDepth = [100, 200, 40];
const [height, , depth] = heightWidthAndDepth;
```

Naming Object Variables

Object variables refer to an instance of a class. Class names are nouns written with the first letter capitalized, like *Person*, *Account*, or *Task*. Object variable names should contain the related class name: a *person* object of the *Person* class, an *account* object of the *Account* class, etc. You can freely decorate the object's name, for example, with an adjective: *completedTask*. It is important to include the class name or at least some significant part of it at the end of the variable name. Then looking at the end of the variable name tells what kind of object is in question.

Sometimes you might want to name an object variable so that the name of its class is implicit, for example:

```
// The class of the function parameters, 'Location', is implicit
drive(home, destination);
```

In the above example, the classes of `home` and `destination` objects are not explicit. In most cases, it is preferable to make the class name explicit in the variable name when it does not make the variable name too long. This is because of the variable type deduction. The types of variables are not necessarily visible in the code, so the type of a variable should be communicated by the variable name. Below is an example where the types of function parameters are explicit.

```
// The class of the function parameters, 'Location', is now explicit
drive(homeLocation, destLocation);
```

Naming Optional Variables

How to name optional variables depends on the programming language and how the optional types are implemented. An optional variable name should be prefixed with *maybe* in languages where you need to unwrap the possible value from an optional object.

In Java, when using `Optional<T>`, name variables of this type using the following pattern: *maybe<Something>*:

```
maybeLoggedInUser.ifPresent(loggedInUser -> loggedInUser.logout());
final User currentUser = maybeLoggedInUser.orElse(guestUser);
```

In TypeScript and other languages where optional types are created using type unions, you don't need any prefixes in optional variable names. In the below example, the `discount` parameter is optional, and its type is `number | undefined`:

```
function addTax(
  price: number,
  discount?: number
): number {
  return 1.2 * (price — (discount ?? 0));
}

const priceWithTax = addTax(priceWithoutTax);
```

Naming Function Variables (Callbacks)

Callback functions are functions supplied to other functions to be called at some point. If a callback function returns a value, it can be named according to the returned value, but it should still contain a verb. If the callback function does not return a value, you should name the callback function like any other function: Indicating what the function does. Suppose you have a variable storing a function object, like a Java `Function` instance. In that case, you need to name the variable according to the rules for an object variable, i.e., the variable name should be a noun. For example, if you have a Java `Function` object currently named `map`, you should correct the name to be a noun, like `mapper`.

```
const doubledValue = value => 2 * value;
const squaredValue = value => value * value;
const valueIsEven = nbr => (nbr % 2) === 0;
const values = [1, 2, 3, 4, 5]
const doubledValues = values.map(doubledValue);
const squaredValues = values.map(squaredValue);
const evenValues = values.filter(valueIsEven);

const strings = [" string1", "string2 "];
const trimmedString = str => str.trim();
const trimmedStrings = strings.map(trimmedString);

const sumOfValues = (sum, value) => sum + value;
```

```
values.reduce(sumOfValues, 0);
```

The above example would be even more apparent if the `map` function was renamed to `mapEach` and the `reduce` function was renamed to `reduceTo`:

```
const doubledValues = values.mapEach(doubledValue);
const trimmedStrings = strings.mapEach(trimmedString);
values.reduceTo(sumOfValues, 0);
```

Let's have an example written in Clojure:

```
(defn print-first-n-doubled-integers [n]
  (println (take n (map (fn [x] (* 2 x)) (range)))))
```

To understand what happens in the above code, you should start reading from the innermost function call and proceed toward the outermost function call. When traversing the function call hierarchy, the difficulty lies in storing and retaining information about all the nested function calls in short-term memory.

We could simplify reading the above example by giving a name to the anonymous function and introducing variables (constants) for intermediate function call results. Of course, our code becomes more prolonged, but coding is not a competition to write the shortest possible code but to write the shortest, most readable, and understandable code for other people and your future self. It is a compiler's job to compile the below longer code into as efficient code as the above shorter code.

Below is the above code refactored:

```
(defn print-first-n-doubled-integers [n]
  (let [doubled (fn [x] (* 2 x))
        doubled-integers (map doubled (range))
        first-n-doubled-integers (take n doubled-integers)]
          (println first-n-doubled-integers)))
```

Let's think hypothetically: if Clojure's `map` function took parameters in a different order and the `range` function was named `integers` (to better describe what it returns) and the `take` function was named `take-first` (like `take-last`), we would have an even more explicit version of the original code:

```
(defn print-first-n-doubled-integers [n]
  (let [doubled (fn [x] (* 2 x))
        doubled-integers (map (integers) doubled)
        first-n-doubled-integers (take-first n doubled-integers)]
          (println first-n-doubled-integers)))
```

Naming Class Properties

Class properties (i.e., class attributes, fields, or member variables) should be named so that the class name is not repeated in the property names. Below is an example of incorrect naming:

```
public class Order {
  private long orderId;
  private OrderState orderState;
}
```

Below is the above code with corrected names:

```
public class Order {
  private long id;
  private OrderState state;
}
```

If you have a class property to store a callback function (e.g., event handler or lifecycle callback), you should name it so that it tells on what occasion the stored callback function is called. Name properties storing event handlers using the following pattern: *on* + *<event-type>*, e.g., *onClick* or *onSubmit*. Name properties storing lifecycle callbacks in a similar way you would name a lifecycle method, for example: *onInit*, *afterMount*, or *beforeMount*.

General Naming Rules

Use Short, Common Names

When picking a name for something, use the most common shortest name. If you have a function named *relinquishSomething*, consider a shorter and more common name for the function. You could rename the function to *releaseSomething*, for example. The word "release" is shorter and more common than the "relinquish" word. Use Google to search for word synonyms, e.g., "relinquish synonym", to find the shortest and most common similar term.

Pick One Name And Use It Consistently

Let's assume that you are building a data exporter microservice and you are currently using the following terms in the code: *message*, *report*, *record* and *data*. Instead of using four different terms to describe the same thing, you should pick just one term, like *message*, for example, and use it consistently throughout the microservice code.

Suppose you need to figure out a term to indicate a property of a class. You should pick just one term, like *property*, and use it consistently everywhere. You should not use multiple terms like *attribute*, *field*, and *member* to describe a property of a class.

Avoid Obscure Abbreviations

Many abbreviations are commonly used, like *str* for a string, *num/nbr* for a number, *prop* for a property, or *val* for a value. Most programmers use these, and I use them to make long names shorter. If a variable name is short, the full name should be used instead, like *numberOfItems* instead of *nbrOfItems*. Use abbreviations in cases where the variable name otherwise becomes too long. What I especially try to avoid is using uncommon abbreviations. For example, I would never abbreviate *amount* to *amnt* or *discount* to *dscnt* because I haven't seen those abbreviations used much in real life.

Avoid Too Short Or Meaningless Names

Names that are too short do not communicate what the variable is about. As loop counters, use a variable name like *index* or *<something>Index* if the loop variable is used to index something, like an array, for example. An indexing variable should start from zero. If the loop variable is counting the number of things, use *number* or *<something>Number* as the variable name, and start the loop counter from value one instead of zero. For example, a loop to start five threads should be written in C++ in the following way:

```
for (size_t threadNumber{1U}; threadNumber <= 5U; ++threadNumber)
{
  startThread(threadNumber);
}
```

If you don't need to use the loop counter value inside the loop, you can use a loop variable named *count:*

```
for (size_t count{1U}; count <= objectCount; ++count)
{
  objects.push_back(acquireObject(std::forward<Args>(args)...));
}
```

Uniform Source Code Repository Structure Principle

> *Structuring code in a source code repository systematically in a certain way makes it easy for other developers to discover wanted information quickly.*

Below are examples of ways to structure source code repositories for Java, C++, and JavaScript/TypeScript microservices. In the below examples, a containerized (Docker) microservice deployed to a Kubernetes cluster is assumed. Your CI tool might require that the CI/CD pipeline code must reside in a specific directory. But if not, place the CI/CD pipeline code in a *ci-cd* directory.

Java Source Code Repository Structure

Below is the proposed source code repository structure for a Java microservice (Gradle build tool is used):

```
java-service
├── ci-cd
│   └── Jenkinsfile
├── docker
│   ├── Dockerfile
│   └── docker-compose.yml
├── docs
├── env
│   ├── .env.dev
│   └── .env.ci
├── gradle
│   └── wrapper
│       └── ...
├── helm
│   └── java-service
│       ├── templates
│       ├── .helmignore
│       ├── Chart.yaml
│       ├── values.schema.json
│       └── values.yaml
├── integration-tests
│   ├── features
│   │   └── feature1.feature
│   └── steps
├── scripts
│   └── // Bash scripts here...
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com.domain.java-service
│   │   │       └── // source code
│   │   └── resources
│   └── test
│       ├── java
│       │   └── com.domain.java-service
│       │       └── // unit test code
│       └── resources
├── .gitignore
├── build.gradle
├── gradlew
├── gradlew.bat
├── README.MD
└── settings.gradle
```

C++ Source Code Repository Structure

Below is the proposed source code repository structure for a C++ microservice (CMake build tool is used):

```
cpp-service
├── ci-cd
│   └── Jenkinsfile
├── docker
│   ├── Dockerfile
│   └── docker-compose.yml
├── docs
├── env
│   ├── .env.dev
│   └── .env.ci
├── helm
│   └── cpp-service
│       ├── templates
│       ├── .helmignore
│       ├── Chart.yaml
│       ├── values.schema.json
│       └── values.yaml
├── integration-tests
│   ├── features
│   │   └── feature1.feature
│   └── steps
├── scripts
│   └── // Bash scripts here...
├── src
│   ├── // source code here
│   │   main.cpp
│   └── CMakeLists.txt
├── test
│   ├── // unit test code
│   │   main.cpp
│   └── CMakeLists.txt
├── .gitignore
├── CMakeLists.txt
└── README.MD
```

JavaScript/TypeScript Source Code Repository Structure

Below is the proposed source code repository structure for a JavaScript/TypeScript microservice:

```
ts-service
├── ci-cd
│   └── Jenkinsfile
├── docker
│   ├── Dockerfile
│   └── docker-compose.yml
├── docs
├── env
│   ├── .env.dev
│   └── .env.ci
├── helm
│   └── ts-service
│       ├── templates
│       ├── .helmignore
│       ├── Chart.yaml
│       ├── values.schema.json
│       └── values.yaml
├── integration-tests
│   ├── features
│   │   └── feature1.feature
│   └── steps
```

```
├── scripts
│   └── // Bash scripts here...
├── src
│   └── // source code here
├── test
│   └── // unit test code here
├── .gitignore
├── .eslintrc.json
├── .prettier.rc
├── package.json
├── package-lock.json
├── README.MD
└── tsconfig.json
```

Domain-Based Source Code Structure Principle

> *Structure source code tree primarily by domains, not by technical details. Each source code directory should have a single responsibility at its abstraction level.*

Below is an example of a Spring Boot microservice's *src* directory that is not organized by domains but is incorrectly organized according to technical details:

```
spring-example-service/
└── src/
    └── java/
        └── com.silensoft.springexampleservice/
            ├── controllers/
            │   ├── AController.java
            │   └── BController.java
            ├── entities/
            │   ├── AEntity.java
            │   └── BEntity.java
            ├── errors/
            │   ├── AError.java
            │   └── BError.java
            ├── dtos/
            │   ├── ADto.java
            │   └── BDto.java
            ├── repositories/
            │   ├── ARepository.java
            │   └── BRepository.java
            └── services/
                ├── AService.java
                └── BService.java
```

Below is the above example modified so that directories are organized by domains:

```
spring-example-service/
└── src/
    └── java/
        └── com.silensoft.springexampleservice/
            ├── domainA/
            │   ├── AController.java
            │   ├── ADto.java
            │   ├── AEntity.java
```

```
            │   ├── AError.java
            │   ├── ARepository.java
            │   └── AService.java
            └── domainB/
                ├── BController.java
                ├── BDto.java
                ├── BEntity.java
                ├── BError.java
                ├── BRepository.java
                └── BService.java
```

You can have several levels of nested domains:

```
spring-example-service/
└── src/
    └── java/
        └── com.silensoft.springexampleservice/
            ├── domainA/
            │   ├── domainA-1/
            │   │   ├── A1Controller.java
            │   │   └── ...
            │   └── domainA-2/
            │       ├── A2Controller.java
            │       └── ...
            └── domainB/
                └── BController.java
```

If you want, you can create subdirectories for technical details inside a domain directory. This is the recommended approach if, otherwise, the domain directory would contain more than 5 to 7 files. Below is an example of the *salesitem* domain:

```
sales-item-service
└── src
    └── java
        └── com.silensoft.salesitemservice
            └── salesitem
                ├── dtos
                │   ├── SalesItemArg.java
                │   └── SalesItemResponse.java
                ├── entities
                │   └── SalesItem.java
                ├── errors
                │   └── SalesItemRelatedError.java
                │   └── SalesItemRelatedError2.java
                ├── repository
                │   └── SalesItemRepository.java
                ├── service
                │   ├── SalesItemService.java
                │   └── SalesItemServiceImpl.java
                └── SalesItemController.java
```

Below is the source code directory structure for the data exporter microservice designed in the previous chapter. There are subdirectories for the four subdomains: input, internal message, transformer, and output. There is a subdirectory created for each common nominator in the class names. It is effortless to navigate the directory tree when locating a particular file. Also, the number of source code files in each directory is low. You can grasp the contents of a directory with a glance.

The problem with directories containing many files is that it is not easy to find the wanted file. For this reason, a single directory should ideally have 2-4 files. The absolute maximum is 5-7 files.

Note that below, a couple of directories are left unexpanded to shorten the example. It should be easy for the reader to infer the contents of the unexpanded directories.

```
src
├── common
├── input
│   ├── config
│   │   ├── parser
│   │   │   ├── json
│   │   │   │   ├── JsonInputConfigParser.cpp
│   │   │   │   └── JsonInputConfigParser.h
│   │   │   └── InputConfigParser.h
│   │   ├── reader
│   │   │   ├── localfilesystem
│   │   │   │   ├── LocalFileSystemInputConfigReader.cpp
│   │   │   │   └── LocalFileSystemInputConfigReader.h
│   │   │   └── InputConfigReader.h
│   │   ├── InputConfig.h
│   │   ├── InputConfigImpl.cpp
│   │   └── InputConfigImpl.h
│   │
│   └── message
│       ├── consumer
│       │   ├── kafka
│       │   │   ├── KafkaInputMessageConsumer.cpp
│       │   │   └── KafkaInputMessageConsumer.h
│       │   └── InputMessageConsumer.h
│       ├── decoder
│       │   ├── avrobinary
│       │   │   ├── AvroBinaryInputMessageDecoder.cpp
│       │   │   └── AvroBinaryInputMessageDecoder.h
│       │   └── InputMessageDecoder.h
│       ├── kafka
│       │   ├── KafkaInputMessage.cpp
│       │   └── KafkaInputMessage.h
│       │
│       └── InputMessage.h
├── internalmessage
│   ├── field
│   ├── InternalMessage.h
│   ├── InternalMessageImpl.cpp
│   └── InternalMessageImpl.h
├── transformer
│   ├── config
│   │   ├── parser
│   │   ├── reader
│   │   ├── TransformerConfig.h
│   │   ├── TransformerConfigImpl.cpp
│   │   └── TransformerConfigImpl.h
│   ├── field
│   │   ├── copy
│   │   │   ├── CopyFieldTransformer.cpp
│   │   │   └── CopyFieldTransformer.h
│   │   ├── expression
│   │   ├── filter
│   │   ├── typeconversion
│   │   ├── FieldTransformer.h
│   │   ├── FieldTransformers.h
```

```
│   │   ├── FieldTransformaresImpl.cpp
│   │   └── FieldTransformersImpl.h
│   └── message
│       ├── MessageTransformer.h
│       ├── MessageTransformerImpl.cpp
│       └── MessageTransformerImpl.h
└── output
    ├── config
    │   ├── parser
    │   ├── reader
    │   ├── OutputConfig.h
    │   ├── OutputConfigImpl.cpp
    │   └── OutputConfigImpl.h
    └── message
        ├── encoder
        │   ├── avrobinary
        │   └── OutputMessageEncoder.h
        ├── producer
        │   ├── pulsar
        │   └── OutputMessageProducer.h
        ├── OutputMessage.h
        ├── OutputMessageImpl.cpp
        └── OutputMessageImpl.h
```

Below is the Java version of the above directory structure:

```
src
├── common
├── input
│   ├── config
│   │   ├── parser
│   │   │   ├── InputConfigParser.java
│   │   │   └── JsonInputConfigParser.java
│   │   ├── reader
│   │   │   ├── InputConfigReader.java
│   │   │   └── LocalFileSystemInputConfigReader.java
│   │   ├── InputConfig.java
│   │   └── InputConfigImpl.java
│   └── message
│       ├── consumer
│       │   ├── InputMessageConsumer.java
│       │   └── KafkaInputMessageConsumer.java
│       ├── decoder
│       │   ├── InputMessageDecoder.java
│       │   └── AvroBinaryInputMessageDecoder.java
│       ├── InputMessage.java
│       └── KafkaInputMessage.java
├── internalmessage
│   ├── field
│   ├── InternalMessage.java
│   └── InternalMessageImpl.java
├── transformer
│   ├── config
│   ├── field
│   │   ├── impl
│   │   │   ├── CopyFieldTransformer.java
│   │   │   ├── ExpressionFieldTransformer.java
│   │   │   ├── FilterFieldTransformer.java
│   │   │   └── TypeConversionFieldTransformer.java
│   │   ├── FieldTransformer.java
│   │   ├── FieldTransformers.java
│   │   └── FieldTransformersImpl.java
```

```
│   └── message
│       ├── MessageTransformer.java
│       └── MessageTransformerImpl.java
└── output
    ├── config
    └── message
```

Below is the source code directory structure for the anomaly detection microservice designed in the previous chapter. The *anomaly* directory is expanded. We can see that our implementation is using JSON for various parsing activities and self-organizing maps (SOM) is used for anomaly detection. JSON and Kafka are used to publish anomaly indicators outside the microservice. Adding new concrete implementations to the below directory structure is straightforward. For example, if we wanted to add YAML support for configuration files, we could create *yaml* subdirectories where we could place YAML-specific implementation classes.

```
src
├── anomaly
│   ├── detection
│   │   ├── configuration
│   │   │   ├── parser
│   │   │   │   ├── json
│   │   │   │   │   ├── JsonAnomalyDetectionConfigParser.cpp
│   │   │   │   │   └── JsonAnomalyDetectionConfigParser.h
│   │   │   │   └── AnomalyDetectionConfigParser.h
│   │   │   ├── AnomalyDetectionConfig.h
│   │   │   ├── AnomalyDetectionConfigFactory.h
│   │   │   ├── AnomalyDetectionConfigFactoryImpl.h
│   │   │   ├── AnomalyDetectionConfigImpl.cpp
│   │   │   └── AnomalyDetectionConfigImpl.h
│   │   ├── engine
│   │   │   ├── AnomalyDetectionEngine.h
│   │   │   ├── AnomalyDetectionEngineImpl.cpp
│   │   │   └── AnomalyDetectionEngineImpl.h
│   │   ├── rule
│   │   │   ├── parser
│   │   │   │   ├── json
│   │   │   │   │   ├── JsonAnomalyDetectionRuleParser.cpp
│   │   │   │   │   └── JsonAnomalyDetectionRuleParser.h
│   │   │   │   └── AnomalyDetectionRuleParser.h
│   │   │   ├── AnomalyDetectionRule.h
│   │   │   ├── AnomalyDetectionRuleFactory.h
│   │   │   ├── AnomalyDetectionRuleFactoryImpl.h
│   │   │   ├── AnomalyDetectionRuleImpl.cpp
│   │   │   └── AnomalyDetectionRuleImpl.h
│   │   ├── AnomalyDetector.h
│   │   ├── AnomalyDetectorImpl.cpp
│   │   └── AnomalyDetectorImpl.h
│   ├── indicator
│   │   ├── publisher
│   │   │   ├── kafka
│   │   │   │   ├── KafkaAnomalyIndicatorPublisher.cpp
│   │   │   │   └── KafkaAnomalyIndicatorPublisher.h
│   │   │   └── AnomalyIndicatorPublisher
│   │   ├── serializer
│   │   │   ├── json
│   │   │   │   ├── JsonAnomalyIndicatorSerializer.cpp
│   │   │   │   └── JsonAnomalyIndicatorSerializer.h
│   │   │   └── AnomalyIndicatorSerializer.h
│   │   ├── AnomalyIndicator.h
```

```
│   │   ├── AnomalyIndicatorFactory.h
│   │   ├── AnomalyIndicatorFactoryImpl.h
│   │   ├── AnomalyIndicatorImpl.cpp
│   │   └── AnomalyIndicatorImpl.h
│   └── model
│       ├── som
│       │   ├── SomAnomalyModel.cpp
│       │   ├── SomAnomalyModel.h
│       │   └── SomAnomalyModelFactory.h
│       ├── training
│       │   ├── engine
│       │   │   ├── AnomalyModelTrainingEngine.h
│       │   │   ├── AnomalyModelTrainingEngineImpl.cpp
│       │   │   └── AnomalyModelTrainingEngineImpl.h
│       │   ├── som
│       │   │   ├── SomAnomalyModelTrainer.cpp
│       │   │   └── SomAnomalyModelTrainer.h
│       │   └── AnomalyModelTrainer.h
│       ├── AnomalyModel.h
│       └── AnomalyModelFactory.h
├── common
├── measurement
├── Application.h
├── Application.cpp
├── DependencyInjector.h
└── main.cpp
```

Let's have one more example with a *data-visualization-web-client*.

This web client's UI consists of the following pages, which all include a common header:

- Dashboards
- Data Explorer
- Alerts

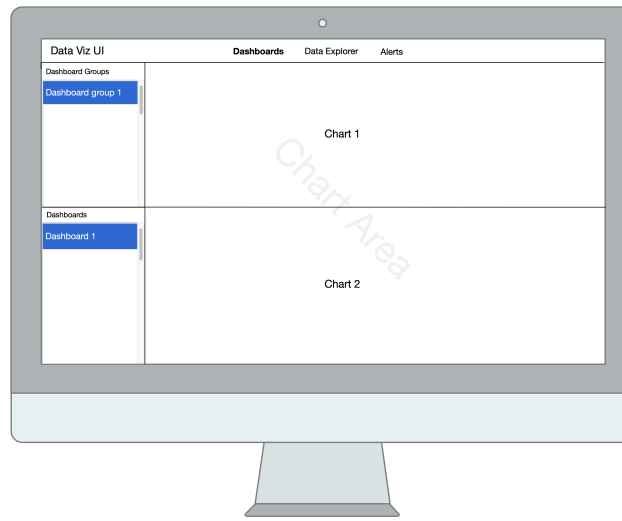

Figure 5.1. Dashboards Page

The *Dashboards* page contains a dashboard group selector, dashboard selector, and chart area to display the selected dashboard's charts. You can select the shown dashboard by first selecting a dashboard group and then a dashboard from that group.

The *Data Explorer* page contains selectors for choosing a data source, measure(s), and dimension(s). The page also contains a chart area to display charts. Using the selectors, a user can change the shown measure(s) and dimension(s) for the currently selected chart in the chart area.

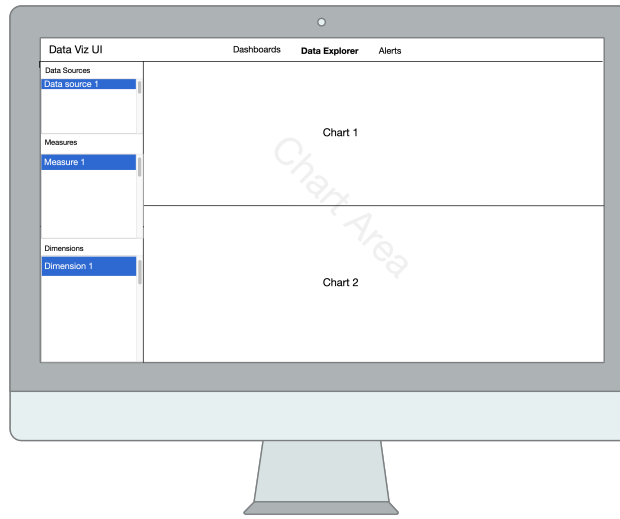

Figure 5.2. Data Explorer Page

Based on the above design, the web client can be divided into the following subdomains:

- Common UI components
    - Chart Area
        - Chart
- Header
- Pages
    - Alerts
    - Dashboards
    - Data Explorer

The source code tree should look like the following:

```
src
├── app
│   ├── common
│   │   └── chartarea
│   │       └── chart
│   ├── header
│   └── pages
│       ├── alerts
│       ├── dashboards
│       │   └── selectors
│       │       ├── dashboardgroup
│       │       └── dashboard
│       └── dataexplorer
│           └── selectors
│               ├── datasource
│               ├── dimension
│               └── measure
├── index.ts
└── store.ts
```

Below is an example of what a single subdomain directory can look like when using React, Redux and SCSS modules:

```
src
├── app
│   └── header
│       ├── model
│       │   ├── actions
│       │   │   ├── AbstractHeaderAction.ts
│       │   │   └── NavigateToPageAction.ts
│       │   ├── services
│       │   └── state
│       │       ├── types
│       │       ├── HeaderState.ts
│       │       └── initialHeaderState.ts
│       ├── view
│       │   ├── navigation
│       │   │   ├── NavigationView.module.scss
│       │   │   └── NavigationView.tsx
│       │   ├── HeaderView.module.scss
│       │   └── HeaderView.tsx
│       └── headerController.ts
├── index.ts
└── store.ts
```

In the above example, we have created two directories for the technical details of the *header* domain: model and view directories. The model directory contains actions, services, and the state, and the view directory contains the view component, its possible subcomponents and CSS definitions. The model's state directory can contain a subdirectory for types used in the subdomain state. The state directory should always contain the type definition for the subdomain's state and the initial state. The services directory contains a service or services that use backend services to control the backend model.

Avoid Comments Principle

Avoid comments in code. The only exception is when documenting the public API of a library.

Comments can be problematic. You cannot trust them 100% because they can be misleading, outdated, or downright wrong. You can only trust the source code itself. Comments are often entirely unnecessary and only make the code more verbose. The following sections describe several ways to avoid writing comments and still keep your code understandable.

Name Things Properly

When you name things like a function poorly, you might end up attaching a comment to the function. To avoid writing comments, it is imperative to focus on naming things correctly. When following the *single responsibility principle* and the *uniform naming principle*, it should be easier to name things correctly and avoid comments. Below is an example of a function with a comment:

MessageBuffer.h

```
class MessageBuffer
{
public:
  // Return false if buffer full,
  // true if message written to buffer
  bool write(const std::shared_ptr<Message>& message);
}
```

If we drop the comment, we will have the following code:

MessageBuffer.h

```
class MessageBuffer
{
public:
  bool write(const std::shared_ptr<Message>& message);
}
```

Dropping the comment alone is not the best solution because some crucial information is now missing. What does that boolean return value mean? It is not 100% clear. We can assume that returning `true` means that message was successfully written, but nothing is communicated about returning `false`. We can only assume it is some error, but not sure what error.

In addition to removing the comment, we should give a better name for the function and rename it as follows:

MessageBuffer.h

```
class MessageBuffer
{
public:
```

```
  bool writeIfBufferNotFull(
    const std::shared_ptr<Message>& message
  );
}
```

Now the purpose of the function is clear, and we can be sure what the boolean return value means. It means whether the message was written to the buffer. Now we also know why the writing of a message can fail: the buffer is full. This will give the function caller sufficient information about what to do next. It should probably wait a while so that the buffer reader has enough time to read messages from the buffer and free up some space.

Below is a real-life example from a book that I once read:

```
public interface Mediator {
  // To register an employee
  void register(Person person);

  // To send a message from one employee to another employee
  void connectEmployees(Person fromPerson,
                        Person toPerson,
                        String msg);

  // To display currently registered members
  void displayDetail();
}
```

There are three functions in the above example, each of which has a problem. The first function is registering a person, but the comment says it is registering an employee. So, there is a mismatch between the comment and the code. In this case, I trust the code over the comment. The correction is to remove the comment because it does not bring any value. It only causes confusion.

The second function says in the comment that it sends a message from one employee to another. The function name tells about connecting employees, but the parameters are persons. I assume that a part of the comment is correct: to send a message from someone to someone else. But once again, I trust the code more over the comment and assume the message is sent from one person to another. We should remove the comment and rename the function.

In the third function, the comment adds information missing from the function name. The comment also discusses members, as other parts of the code speak about employees and persons. There are three different terms used: employee, person, and member. Just one term should be picked. Let's choose the term *person* and use it systematically.

Below is the refactored version without the comments:

```
public interface Mediator {
  void register(Person person);

  void send(String message,
            Person sender,
            Person recipient);
```

```
  void displayDetailsOfRegisteredPersons();
}
```

Single Return Of Named Value At The End Of Function

> *A function should have a single return statement and return a named value at the end of the function. Then the code reader can infer the return value meaning by looking at the end of the function.*

Consider the following example:

Metrics.h

```
class Metrics
{
public:
  // ...

  static uint32_t addCounter(
    CounterFamily counterFamily,
    const std::map<std::string, std::string>& labels
  );

  static void incrementCounter(uint32_t counterIndex,
                               size_t incrementAmount);

  // addGauge...
  // setGaugeValue...
}
```

What is the return value of the `addCounter` function? Someone might think a comment is needed to describe the return value because it is unclear what `uint32_t` means. Instead of writing a comment, we can introduce a named value (= variable/constant) to be returned from the function. The idea behind the named return value is that it communicates the semantics of the return value without the need for a comment. In C++, you jump from the function declaration to the function definition to see what the function returns. Below is the implementation for the `addCounter` function:

Metrics.cpp

```
uint32_t Metrics::addCounter(
  const CounterFamily counterFamily,
  const std::map<std::string, std::string>& labels)
{
  uint32_t counterIndex;

  // Perform adding a counter here and
  // set value for the 'counterIndex'

  return counterIndex;
}
```

In the above implementation, we have a single return of a named value at the end of the function.

All we have to do is to look at the end of the function and spot the return statement, which should tell us the meaning of the mysterious `uint32_t` typed return value: It is a counter index. And we can spot that the `increaseCounter` function requires a `counterIndex` argument and this establishes a connection between calling the `addCounter` function first, storing the returned counter index, and later using that stored counter index in calls to the `increaseCounter` function.

Return Type Aliasing

In the previous example, there was the mysterious return value of type `uint32_t` in the `addCounter` function. We learned how introducing a named value returned at the end of the function helped to communicate the semantics of the return value. But there is an even better way to communicate the semantics of a return value. Many languages like C++ and TypeScript offer type aliasing that can be used to communicate the return value semantics. Below is an example where we introduce a `CounterIndex` type alias for the `uint32_t` type:

Metrics.h

```
class Metrics
{
public:
  using CounterIndex = uint32_t;

  // ...

  static CounterIndex addCounter(
    CounterFamily counterFamily,
    const std::map<std::string, std::string>& labels
  );

  static void incrementCounter(CounterIndex counterIndex,
                               size_t incrementAmount);
}
```

And here is the same example in TypeScript:

Metrics.ts

```
export type CounterIndex = number;

export default class Metrics {
  // ...

 static addCounter(
    counterFamily: CounterFamily,
    labels: Record<string, string>
  ): CounterIndex;

  static incrementCounter(counterIndex: CounterIndex,
                          incrementAmount: number): void;
}
```

Some languages, like Java, don't have type aliases. Then you can introduce a wrapper class for the returned value. Here is the same example in Java:

CounterIndex.java

```
public class CounterIndex {
  private final int value;

  public CounterIndex(final int value) {
    this.value = value;
  }

  public int get() {
    return value;
  }
}
```

Metrics.java

```
public final class Metrics {
  // ...

  public static CounterIndex addCounter(
    final CounterFamily counterFamily,
    final Map<String, String> labels
  ) {
    // ...
  }

  public static void incrementCounter(
    CounterIndex counterIndex,
    double incrementAmount
  ) {
    // ...
  }
}
```

We can improve the above example. The `CounterIndex` class could be derived from a generic `Value` class:

```
public class Value<T> {
  private final T value;

  public Value(final T value) {
    this.value = value;
  }

  public T get() {
    return value;
  }
}

public class CounterIndex extends Value<Integer> {
  // ...
}
```

We can improve the above metrics example a lot. First, we should avoid the primitive type obsession. We should not be returning an index from the `addCounter` method, but we should

rename the method as `createCounter` and return an instance of a `Counter` class from the method. Then we should make the example more object-oriented by moving the `incrementCounter` method to the `Counter` class and naming it just `increment`. Also, the name of the `Metrics` class should be changed to `MetricFactory`. And finally, we should make the `MetricFactory` class a singleton instead of containing static methods.

Extract Constant for Boolean Expression

By extracting a constant for a boolean expression, we can eliminate comments. Below is an example where a comment is written below an if-statement and its boolean expression:

MessageBuffer.cpp

```
bool MessageBuffer::writeIfBufferNotFull(
  const std::shared_ptr<Message>& message
) {
  bool messageWasWritten{false};

  if (m_messages.size() < m_maxBufferSize)
  {
    // Buffer is not full
    m_messages.push_back(message);
    messageWasWritten = true;
  }

  return messageWasWritten;
}
```

By introducing a constant to be used in the "buffer is full" check, we can get rid of the "Buffer is not full" comment:

MessageBuffer.cpp

```
bool MessageBuffer::writeIfBufferNotFull(
  const std::shared_ptr<Message> message
) {
  bool messageWasWritten{false};

  const bool bufferIsNotFull =
    m_messages.size() < m_maxBufferSize;

  if (bufferIsNotFull)
  {
    m_messages.push_back(message);
    messageWasWritten = true;
  }

  return messageWasWritten;
}
```

Extract Named Constant or Enumerated Type

If you encounter a *magic number* in your code, you should introduce either a named constant or an enumerated type (enum) for that value. In the below example, we are returning two magic numbers, 0 and 1:

main.cpp

```
int main()
{
  Application application;

  if (application.run())
  {
    // Application was run successfully
    return 0;
  }

  // Exit code: failure
  return 1;
}
```

Let's introduce an enumerated type, `ExitCode`, and use it instead of magic numbers:

main.cpp

```
enum class ExitCode
{
  Success = 0,
  Failure = 1
};

int main()
{
  ExitCode exitCode;
  Application application;
  const bool appWasSuccessfullyRun = application.run();

  if (appWasSuccessfullyRun)
  {
    exitCode = ExitCode::Success;
  }
  else
  {
    exitCode = ExitCode::Failure;
  }

  return static_cast<int>(exitCode);
}
```

It is now easy to add more exit codes with descriptive names later if needed.

Extract Function

If you are planning to write a comment above a piece of code, you should extract that piece of code

to a new function. When you extract a well-named function, you don't need to write that comment. The name of the newly extracted function serves as documentation. Below is an example with some commented code:

MessageBuffer.cpp

```
void MessageBuffer::writeFitting(
  std::deque<std::shared_ptr<Message>>& messages
) {
  if (m_messages.size() + messages.size() <= m_maxBufferSize)
  {
    // All messages fit in buffer
    m_messages.insert(m_messages.end(),
                      messages.begin(),
                      messages.end());

    messages.clear();
  }
  else
  {
    // All messages do not fit, write only messages that fit
    const auto messagesEnd = messages.begin() +
                             m_maxBufferSize -
                             m_messages.size();

    m_messages.insert(m_messages.end(),
                      messages.begin(),
                      messagesEnd);

    messages.erase(messages.begin(), messagesEnd);
  }
}
```

Here is the same code with comments refactored out by extracting two new methods:

MessageBuffer.cpp

```
void MessageBuffer::writeFitting(
  std::deque<std::shared_ptr<Message>>& messages
) {
  const bool allMessagesFit = m_messages.size() +
                              messages.size() <= m_maxBufferSize;

  if (allMessagesFit)
  {
    writeAll(messages)
  }
  else
  {
    writeOnlyFitting(messages);
  }
}

void MessageBuffer::writeAll(
  std::deque<std::shared_ptr<Message>>& messages
) {
  m_messages.insert(m_messages.end(),
                    messages.begin(),
                    messages.end());

  messages.clear();
```

```
}

void MessageBuffer::writeOnlyFitting(
  std::deque<std::shared_ptr<Message>>& messages
) {
  const auto messageCountThatFit = m_maxBufferSize -
                                   m_messages.size();

  const auto messagesEnd = messages.begin() +
                           messageCountThatFit;

  m_messages.insert(m_messages.end(),
                    messages.begin(),
                    messagesEnd);

  messages.erase(messages.begin(), messagesEnd);
}
```

Name Anonymous Function

Anonymous functions are common in functional programming, e.g., when using algorithms like *forEach*, *map*, *filter*, and *reduce*. When an anonymous function is long or complex, you should give it a descriptive name and split it into multiple functions if it is too long. This way, you can eliminate comments.

In the below TypeScript example, we have an anonymous function with a comment:

```
// ...

fs.watchFile('/etc/config/LOG_LEVEL', () => {
  // Update new log level
  try {
    const newLogLevel = fs.readFileSync('/etc/config/LOG_LEVEL',
                                        'utf-8').trim();
    tryValidateLogLevel(newLogLevel);
    process.env.LOG_LEVEL = newLogLevel;
  } catch (error) {
    // ...
  }
});
```

We can refactor the above example so that the comment is removed and the anonymous function is given a name:

```
function updateNewLogLevel() {
  try {
    const newLogLevel = fs.readFileSync('/etc/config/LOG_LEVEL',
                                        'utf-8').trim();
    tryValidateLogLevel(newLogLevel);
    process.env.LOG_LEVEL = newLogLevel;
  } catch (error) {
    // ...
  }
}
```

```
fs.watchFile('/etc/config/LOG_LEVEL', updateNewLogLevel);
```

Avoiding Comments in Bash Shell Scripts

Many programmers, myself included, don't enjoy the mysterious syntax of Linux shell commands and scripts. Even the syntax of the simplest expressions can be hard to understand and remember if you don't work with scripts regularly. Of course, the best thing is to avoid writing complex Linux shell scripts and use a proper programming language like Python instead. But sometimes, performing some actions using a shell script is easier. Because the syntax and commands in shell scripts can be hard to understand, many developers tend to solve the problem by adding comments to scripts.

Next, alternative ways to make scripts more understandable without comments are presented. Let's consider the below example from one real-life script I have bumped into:

```
create_network() {
  #create only if not existing yet
  if [[ -z "$(docker network ls | grep $DOCKER_NETWORK_NAME )" ]];
  then
    echo Creating $DOCKER_NETWORK_NAME
    docker network create $DOCKER_NETWORK_NAME
  else
    echo Network $DOCKER_NETWORK_NAME already exists
  fi
}
```

Below is the same example with the following changes:

- The comment was removed, and the earlier commented expression was moved to a well-named function
- The negation in the expression was removed, and the contents of the *then* and *else*-branches were swapped
- Variable names were made camel case to enhance readability

```
dockerNetworkExists() { [[ -n "$(docker network ls | grep $1 )" ]]; }

createDockerNetwork() {
  if dockerNetworkExists $networkName; then
    echo Docker network $networkName already exists
  else
    echo Creating Docker network $networkName
    docker network create $networkName
  fi
}
```

If your script accepts arguments, give the arguments proper names, for example:

```
dataFilePathName=$1
schemaFilePathName=$2
```

The script reader does not have to remember what `$1` or `$2` means, and you don't have to insert any comments to clarify the meaning of the arguments.

If you have a complex command in a Bash shell script, you should not attach a comment to it but extract a function with a proper name to describe the command.

The below example contains a comment:

```
# Update version in Helm Chart.yaml file
sed -i "s/^version:.*/version: $VERSION/g" helm/service/Chart.yaml
```

Here is the above example refactored to contain a function

```
updateHelmChartVersionInChartYamlFile() {
  sed -i "s/^version:.*/version: $1/g" helm/service/Chart.yaml
}

updateHelmChartVersionInChartYamlFile $version
```

Here is another example:

```
getFileLongestLineLength() {
  echo $(awk '{ if (length($0) > max) max = length($0) } END { print max }' $1)
}

configFileLongestLineLength = $(getFileLongestLineLength $configFilePathName)
```

Function Single Return Principle

> *Prefer a single return statement at the end of a function to clearly communicate the return value's meaning and make refactoring the function easier.*

A single return statement with a named value at the end of a function clearly communicates the return value semantics if the return value type does not directly communicate it. For example, if you return a value of a primitive type like an integer or boolean from a function, it is not necessarily 100% clear what the return value means. But when you return a named value at the end of the function, the name of the returned variable communicates the semantics.

You might think that being unable to return a value in the middle of a function would make the function less readable because of lots of nested if-statements. This is possible, but one should remember that a function should be small. Aim to have a maximum of 5-9 lines of statements in a single function. Following that rule, you never have *a hell of nested if-statements* inside a single function.

Having a single return statement at the end of a function makes refactoring the function easier. You

can use automated refactoring tools provided by your IDE. It is always harder to extract a new function from code containing a return statement. The same is true for loops with a *break* or *continue* statement. It is easier to refactor code inside a loop that does not contain a break or continue statement.

In some cases, returning a single value at the end of a function makes the code more straightforward and requires fewer lines of code.

Below is an example of a function with two return locations:

TransformThread.cpp

```
bool TransformThread::transform(
  const std::shared_ptr<InputMessage>& inputMessage
) {
  auto outputMessage = m_outputMessagePool->acquireMessage();
  bool messageIsFilteredIn;

  const bool messageWasTransformed =
    m_messageTranformer->transform(inputMessage,
                                   outputMessage,
                                   messageIsFilteredIn);

  if (messageWasTransformed && messageIsFilteredIn)
  {
    m_outputMessages.push_back(outputMessage);
  }
  else
  {
    m_outputMessagePool->returnMessage(outputMessage);

    if (!messageWasTransformed)
    {
        return false;
    }
  }

  return true;
}
```

When analyzing the above function, we notice that it transforms an input message into an output message. We can conclude that the function returns *true* on successful message transformation. We can shorten the function by refactoring it to contain only one return statement. After refactoring, it is 100% clear what the function return value means.

TransformThread.cpp

```
bool TransformThread::transforme(
  const std::shared_ptr<InputMessage>& inputMessage
) {
  auto outputMessage = m_outputMessagePool->acquireMessage();
  bool messageIsFilteredIn;

  const bool messageWasTransformed =
      m_messageTransformer->transform(inputMessage,
                                      outputMessage,
                                      messageIsFilteredIn);
```

```
  if (messageWasTransformed && messageIsFilteredIn)
  {
    m_outputMessages.push_back(outputMessage);
  }
  else
  {
    m_outputMessagePool->return(outputMessage);
  }

  return messageWasTransformed;
}
```

As an exception to this rule, you can have multiple return statements in a function when the function has optimal length and would become too long if it is refactored to contain a single return statement. Additionally, it is required that the semantic meaning of the return value is clear from the function name or the return type. Below is an example of a function with multiple return statements. It is also clear from the function name what the return value means. Also, the length of the function is optimal: seven statements.

```
private areEqual(
  iterator: MyIterator<T>,
  anotherIterator: MyIterator<T>
): boolean {
  while (iterator.hasNextElement()) {
    if (anotherCollectionIterator.hasNextElement()) {
      if (iterator.getNextElement() !==
          anotherCollectionIterator.getNextElement()) {
       return false;
      }
    } else {
      return false;
    }
  }

  return true;
}
```

If we refactored the above code to contain a single return statement, the code would become too long (10 statements) to fit in one function, as shown below. In this case, we should prefer the above code over the below code.

```
private areEqual(
  iterator: MyIterator<T>,
  anotherIterator: MyIterator<T>
): boolean {
  let areEqual = true;

  while (iterator.hasNextElement()) {
    if (anotherCollectionIterator.hasNextElement()) {
      if (iterator.getNextElement() !==
          anotherCollectionIterator.getNextElement()) {
       areEqual = false;
       break;
      }
    } else {
      areEqual = false;
```

```
      break;
    }
  }

  return areEqual;
}
```

As the second exception to this rule, you can use multiple return locations in a factory because you know from the factory name what type of objects it creates. Below is an example factory with multiple return statements:

```
enum class CarType
{
  Audi,
  Bmw,
  MercedesBenz
};

class Car
{
  // ...
};

class Audi : public Car
{
  // ...
};

class Bmw : public Car
{
  // ...
};

class MercedesBenz : public Car
{
  // ...
};

class CarFactory
{
public:
  std::shared_ptr<Car> createCar(const CarType carType)
  {
    switch(carType)
    {
      case CarType::Audi:
        return std::make_shared<Audi>();
      case CarType::Bmw:
        return std::make_shared<Bmw>();
      case CarType::MercedesBenz:
        return std::make_shared<MercedesBenz>();
      default:
        throw std::invalid_argument("Unknown car type");
    }
  }
};
```

Prefer a Statically Typed Language for Production Code Principle

> *Prefer a statically typed language when implementing production software. You can use an untyped language like Python for non-production code like integration, end-to-end and automated non-functional tests. And you can use Bash shell scripting for small scripts.*

You can manage with a small software component without types, but when it grows bigger and more people are working with it, the benefits of static typing become evident.

Let's analyze what potential problems using an untyped language might incur:

- Function arguments might be given in the wrong order
- Function argument might be given with the wrong type
- Not all function arguments are given (applicable in some languages)
- Function return value type might be misunderstood
- Forced to write public API comments to describe function signatures
- Type errors are not necessarily found in testing

Function Arguments Might Be Given in Wrong Order

When using an untyped language, you can give arguments to a function in the wrong order if the arguments are of the same type. You won't get a compilation error from this mistake. Modern IDEs can display inlay parameter hints for a function call. This is a feature you should consider enabling in your IDE. Those parameter hints might reveal cases where arguments for a function are not given in the correct order.

Function Argument Might Be Given with Wrong Type

When using an untyped language, you can give a function argument with the wrong type. For example, a function requires a string representation of a number, but you provide a number. Properly naming function arguments can help. Instead of naming a string argument, *amount*, the argument should be named as *amountString* or *amountAsString*.

Not All Function Arguments Are Given

In some languages, like JavaScript, you can give fewer arguments than expected to a function when you call it. This results in having *undefined* values for the arguments not given in the function call. You don't get an error from calling a function with too few arguments.

Function Return Value Type Might Be Misunderstood

Determining the function return value type can be difficult. It is not necessarily 100% clear from the name of the function. For example, if you have a function named `getValue`, it is not 100% clear what the return value type is. It might be apparent only if you know the context of the function well. As an improvement, the function should be appropriately named, for example: `getValueAsString()`, if the returned value is always a string. If the return value type is unclear from the function name, you must analyze the function's source code to determine the return value type. That is unnecessary and error-prone manual work that can be avoided using a typed language.

Forced to Write Public API Comments

When using an untyped language, you might be forced to document a public API using comments. This is additional work that could be avoided by using static types. Writing API documentation with comments is error-prone. You can accidentally write wrong information in the API documentation or forget to update the documentation when you make changes to the API. Similarly, the API documentation readers can make mistakes. They might not read the API documentation at all. Or they have read it earlier but later misremember it.

Type Errors Are Not Found in Testing

This is the biggest problem. You might think that if you have mistakes in your code related to having correct function arguments with the correct types, testing will reveal those mistakes. This is typically a wrong assumption. Unit testing won't find the issues because you mock other classes. You can only find the issues in integration testing when you integrate the software component (i.e., test functions calling other real functions instead of mocks). According to the testing pyramid, integration tests only cover a subset of the codebase, less than unit tests. And depending on the integration testing code coverage, some function argument order or argument/return value type correctness issues may be left untested and escape to production.

Refactoring Principle

> *You cannot write the perfect code on the first try, so you should always reserve some time for future refactoring.*

You need to refactor even if you are writing code for a new software component. Refactoring is not related to legacy codebases only. If you don't refactor, you let technical debt grow in the software. The main idea behind refactoring is that no one can write the perfect code on the first try. Refactoring means that you change code without changing the actual functionality. After

refactoring, most of the tests should still pass, the code is organized differently, and you have a better object-oriented design and improved naming of things. Refactoring does not usually affect integration tests but can affect unit tests depending on the type and scale of refactoring. Keep this in mind when estimating refactoring effort.

We don't necessarily reserve any or enough time for refactoring when we plan things. When we provide work estimates for epics, features, and user stories, we should be conscious of the need to refactor and add some extra time to our initial work estimates (which don't include refactoring). Refactoring is work that is not necessarily understood clearly by the management. The management should support the need to refactor even if it does not bring clear added value to an end user. But it brings value by not letting the codebase rot and removing technical debt. If you have software with lots of accumulated technical debt, it is costly to develop new features and maintain the software. Also, the quality of the software is lower, which can manifest in many bugs and lowered customer satisfaction.

Below is a list of the most common code smells and refactoring techniques to solve them:

Code Smell	Refactoring Solution
Non-descriptive name	Rename
Long method	Extract method
Complex expression	Extract constant
Long switch-case or if-then-else statement	Replace conditionals with polymorphism
Long parameter list	Introduce parameter object
Shotgun surgery	Replace conditionals with polymorphism
Negated boolean condition	Invert If statement

Rename

This is probably the single most used refactoring technique. You often don't get the names right on the first try and need to do renaming. Modern IDEs offer tools that help rename things in the code: interfaces, classes, functions, and variables. The IDE's renaming functionality is always better than the plain old search-and-replace method. If using the search-and-replace method, you can accidentally rename something that is not wanted to be renamed or don't rename something that should have been renamed.

Extract Method

This is probably the second most used refactoring technique. When you implement a public method of a class, the method quickly grows in the number of code lines. A function should contain a maximum of 5-9 statements to keep it readable and understandable. When a public method is too long, you should extract one or more private methods and call these private methods from the public method. Every modern IDE has an *extract method* refactoring tool that allows you to extract private methods easily. Select the code lines you want to extract to a new method and press the IDE's shortcut key for the extract method functionality. Then give a descriptive name for the extracted method, and you are done. In some cases, the refactoring is not automatic. For example, if the code to be extracted contains a *return*, *break*, or *continue* statement that affects the execution flow of the function (causing multiple return points). If you want to keep your code refactorable, avoid using *break*, and *continue* statements and have only a single return statement at the end of the function. You can organize the arguments of the extracted method in better order before completing the extraction in the IDE.

Extract Constant

If you have a complex expression (boolean or numeric), assign the value of the expression to a constant. The name of the constant conveys information about the expression. Below is an example where we make the if-statements read better by extracting expressions to constants:

```
// ...

if (dataSourceSelectorIsOpen &&
    measureSelectorIsOpen &&
    dimensionSelectorIsOpen
) {
  dataSourceSelectorContentElem.style.height =
    `${0.2 * availableHeight}px`;

  measureSelectorContentElem.style.height =
    `${0.4 * availableHeight}px`;

  dimensionSelectorContentElem.style.height =
    `${0.4 * availableHeight}px`;
} else if (!dataSourceSelectorIsOpen &&
           !measureSelectorIsOpen &&
           dimensionSelectorIsOpen
) {
    dimensionSelectorContentElem.style.height
      = `${availableHeight}px`;
}
```

Let's extract constants:

```
// ...

const allSelectorsAreOpen = dataSourceSelectorIsOpen &&
                            measureSelectorIsOpen &&
```

```
                            dimensionSelectorIsOpen;

const onlyDimensionSelectorIsOpen =
    !dataSourceSelectorIsOpen &&
    !measureSelectorIsOpen &&
    dimensionSelectorIsOpen;

if (allSelectorsAreOpen) {
  dataSourceSelectorContentElem.style.height =
    `${0.2 * availableHeight}px`;

  measureSelectorContentElem.style.height =
    `${0.4 * availableHeight}px`;

  dimensionSelectorContentElem.style.height =
    `${0.4 * availableHeight}px`;
} else if (onlyDimensionSelectorIsOpen) {
  dimensionSelectorContentElem.style.height =
    `${availableHeight}px`;
}
```

Below is an example in C++ where we return a boolean expression:

```
bool AvroFieldSchema::equals(
  const std::shared_ptr<AvroFieldSchema>& otherAvroFieldSchema
) const
{
  return m_type == otherAvroFieldSchema->getType() &&
         m_name.substr(m_name.find_first_of('.') + 1U) ==
         otherField->getName().substr(
          otherField->getName().find_first_of('.') + 1U);
}
```

It can be challenging to understand what the boolean expression means. We could improve the function by adding a comment: (We assume that each field name has a root namespace that cannot contain a dot character)

```
bool AvroFieldSchema::equals(
  const std::shared_ptr<AvroFieldSchema>& otherAvroFieldSchema
) const
{
  // Field schemas are equal if field types are equal and
  // field names without the root namespace are equal
  return m_type == otherAvroFieldSchema->getType() &&
         m_name.substr(m_name.find_first_of('.') + 1U) ==
         otherField->getName().substr(
          otherField->getName().find_first_of('.') + 1U);
}
```

But we should not write comments because comments are never 100% trustworthy. It is possible that a comment and the related code are not in synchrony: someone has changed the function without updating the comment or modified only the comment but did not change the function. Let's refactor the above example by removing the comment and extracting multiple constants. The below function is longer than the original, but it is, of course, more readable. If you look at the last two statements of the method, you can understand in what case two field schemas are equal. It

should be the compiler's job to make the below longer version of the function as performant as the original function.

```
bool AvroFieldSchema::equals(
  const std::shared_ptr<AvroFieldSchema>& otherAvroFieldSchema
) const
{
  const auto fieldNameWithoutRootNamespace =
    m_name.substr(m_name.find_first_of('.') + 1U);

  const auto otherFieldName = otherAvroFieldSchema->getName();

  const auto otherFieldNameWithoutRootNamespace =
    otherFieldName.substr(otherFieldName.find_first_of('.') + 1U);

  const bool fieldTypesAndNamesWithoutRootNsAreEqual =
    m_type == otherAvroFieldSchema->getType() &&
    fieldNameWithoutRootNamespace == otherFieldNameWithoutRootNamespace;

  const bool fieldSchemasAreEqual =
    fieldTypesAndNamesWithoutRootNsAreEqual;

  return fieldSchemasAreEqual;
}
```

Replace Conditionals with Polymorphism

Suppose you encounter a large switch-case statement or if/else-if structure in your code (not considering code in factories). It means your software component does not have a proper object-oriented design. You should replace the conditionals with polymorphism. When you introduce proper OOD in your software component, you move the functionality from a switch statement's case branches to different classes that implement a particular interface. And similarly, you move the code from if and else-if statements to different classes that implement a certain interface. This way, you can eliminate the switch-case and if/else-if statements and replace them with a polymorphic method call.

Below is a TypeScript example of non-object-oriented design:

```
function doSomethingWith(chart: Chart) {
  if (chart.getType() === 'column') {
    // do this
  } else if (chart.getType() === 'pie') {
    // do that
  } else if (chart.getType() === 'geographic-map') {
    // do a third thing
  }
}
```

Let's replace the above conditionals with polymorphism:

```
interface Chart {
  doSomething(...): void;
}
```

```
class ColumnChart implements Chart {
  doSomething(...): void {
    // do this
  }
}

class PieChart implements Chart {
  doSomething(...): void {
    // do that
  }
}

class GeographicMapChart implements Chart {
  doSomething(...): void {
    // do a third thing
  }
}

function doSomethingWith(chart: Chart) {
  chart.doSomething();
}
```

Suppose you are implementing a data visualization application and have many places in your code where you check the chart type and need to introduce a new chart type. It could mean you must add a new *case* or *else-if*-statement in many places in the code. This approach is very error-prone and is called *shotgun surgery* because you need to find all the places in the codebase where code needs to be modified. What you should do is conduct proper object-oriented design and introduce a new chart class containing the new functionality instead of introducing that new functionality by modifying code in multiple places.

Introduce Parameter Object

If you have more than 5-7 parameters for a function, you should introduce a parameter object to reduce the number of parameters to keep the function signature more readable. Below is an example constructor with too many parameters:

KafkaConsumer.java

```
public class KafkaConsumer {
  public KafkaConsumer(
    final List<String> brokers,
    final List<String> topics,
    final List<String> extraConfigEntries,
    final boolean tlsIsUsed,
    final boolean certShouldBeVerified,
    final String caFilePathName,
    final String certFilePathName,
    final String keyFilePathName)
  {
    // ...
  }
}
```

Let's group the Transport Layer Security (TLS) related parameters to a parameter class named `TlsOptions`:

TlsOptions.java

```
public class TlsOptions {
  public TlsOptions(
    final boolean tlsIsUsed,
    final boolean certShouldBeVerified,
    final String caFilePathName,
    final String certFilePathName,
    final String keyFilePathName
  ) {
    // ...
  }
}
```

Now we can modify the `KafkaConsumer` constructor to utilize the `TlsOptions` parameter class:

KafkaConsumer.java

```
public class KafkaConsumer {
  public KafkaConsumer(
    final List<String> brokers,
    final List<String> topics,
    final List<String> extraConfigEntries,
    final TlsOptions tlsOptions
  ) {
    // ...
  }
}
```

Invert If Statement

This is a refactoring that a modern IDE can do for you.

Below is a Python example with a negated boolean expression in the if-statement condition. Notice how difficult the boolean expression reads: "hostMountFolder *is not None*". It is a double-negative statement and thus difficult to read.

```
def getBehaveTestFolder(relativeTestFolder = ''):
  hostMountFolder = os.environ.get("HOST_MOUNT_FOLDER")

  if hostMountFolder is not None:
    finalHostMountFolder = hostMountFolder
    if hostMountFolder.startswith("/mnt/c/"):
      finalHostMountFolder = hostMountFolder.replace("/mnt/c/", \
                                                     "/c/", 1)

    behaveTestFolder = finalHostMountFolder + '/' + \
                       relativeTestFolder
  else:
    behaveTestFolder = os.getcwd()

  return behaveTestFolder
```

Let's refactor the above code so that the if and else statements are inverted:

```
def getBehaveTestFolder(relativeTestFolder = ''):
  hostMountFolder = os.environ.get("HOST_MOUNT_FOLDER")

  if hostMountFolder is None:
    behaveTestFolder = os.getcwd()
  else:
    finalHostMountFolder = hostMountFolder
    if hostMountFolder.startswith("/mnt/c/"):
      finalHostMountFolder = hostMountFolder.replace("/mnt/c/", \
                                                     "/c/", 1)
    behaveTestFolder = finalHostMountFolder + '/' + \
                       relativeTestFolder

  return behaveTestFolder
```

Below is another example in C++:

```
if (somePointer != nullptr)
{
  // Do thing 1
}
else
{
  // Do thing 2
}
```

We should not have a negation in the if-statement's condition. Let's refactor the above example:

```
if (somePointer == nullptr)
{
  // Do thing 2
}
else
{
  // Do thing 1
}
```

Static Code Analysis Principle

> *Let the computer find bugs and issues in the code for you.*

Static code analysis tools find bugs and design-related issues on your behalf. Use multiple static code analysis tools to get the full benefit. Different tools might detect different issues. Using static code analysis tools frees people's time in code reviews to focus on things that automation cannot tackle.

Below is a list of some common static code analysis tools for different languages:

- Java
    - Jetbrains IntelliJ IDEA IDE inspections
    - SonarLint
    - SonarQube/SonarCloud
- C++
    - Jetbrains CLion IDE inspections
    - Clang-Tidy
    - MISRA C++ 2008 guidelines
    - CppCheck
    - SonarLint
    - SonarQube/SonarCloud
- TypeScript
    - Jetbrains WebStorm IDE inspections
    - ESLint (+ various plugins, like TypeScript plugin)
    - SonarLint
    - SonarQube/SonarCloud

Infrastructure and deployment code should be treated the same way as source code. Remember to run static code analysis tools on your infrastructure and deployment code, too. Several tools are available for analyzing infrastructure and deployment code, like *Checkcov*, which can be used for analyzing Terraform, Kubernetes, and Helm code. Helm tool contains a linting command to analyze Helm chart files, and *Hadolint* is a tool for analyzing *Dockerfiles* statically.

Common Static Code Analysis Issues

Issue	Description/Solution
Chain of instance of checks	This issue indicates a chain of conditionals in favor of object-oriented design. Use the *replace conditionals with polymorphism* refactoring technique to solve this issue.
Feature envy	Use the *don't ask, tell principle* from the previous chapter to solve this issue.
Use of concrete classes	Use the *program against interfaces* principle from the previous chapter to solve this issue.
Assignment to a function argument	Don't modify function arguments but introduce a new variable. You can avoid this issue in Java by declaring function parameters as *final*.

Issue	Description/Solution
Commented-out code	Remove the commented-out code. If you need that piece of code in the future, it is available in the version control system forever.
Const correctness	Make variables and parameters const or final whenever possible to achieve immutability and avoid accidental modifications.
Nested switch statement	Use switch statements mainly only in factories. Do not nest them.
Nested conditional expression	Conditional expression (?:) should not be nested because it greatly hinders the code readability.
Overly complex boolean expression	Split the boolean expression into parts and introduce constants to store the parts and the final expression.
Expression can be simplified	This can be refactored automatically by the IDE.
Switch statement without default branch	Always introduce a default branch and throw an exception from there. Otherwise, when you are using a switch statement with an enum, you might encounter strange problems after adding a new enum value that is not handled by the switch statement.
Law of Demeter	The object knows too much. It is coupled to the dependencies of another object, which creates additional coupling and makes code harder to change.
Reuse of local variable	Instead of reusing a variable for a different purpose, introduce a new variable. That new variable can be named appropriately to describe its purpose.
Scope of variable is too broad	Introduce a variable only just before it is needed.
Protected field	Subclasses can modify the protected state of the superclass without the superclass being able to control that. This is an indication of breaking the encapsulation and should be avoided.
Breaking the encapsulation: Return of modifiable/ mutable field	Use the *Don't leak modifiable internal state outside an object principle* from the previous chapter to solve this issue.
Breaking the encapsulation: Assignment from a method parameter to a modifiable/mutable field	Use the *Don't assign from a method parameter to a modifiable field principle* from the previous chapter to solve this issue.
Non-constant public field	Anyone can modify a public field. This breaks the encapsulation and should be avoided.

Issue	Description/Solution
Overly broad catch-block	This can indicate a wrong design. Don't catch the language's base exception class if you should only catch your application's base error class, for example. Read more about handling exceptions in the next section.

Error/Exception Handling Principle

Many languages like C++, Java, and TypeScript have an exception-handling mechanism that can handle errors and exceptional situations. First of all, I want to make a clear distinction between these two words:

> *An **error** is something that can happen, and one should be prepared for it. An **exception** is something that should never happen.*

You define errors in your code and raise them in your functions. For example, if you try to write to a file, you must be prepared for the error that the disk is full, or if you are reading a file, you must be prepared for the error that the file does not exist (anymore).

Many errors are recoverable. You can delete files from the disk to free up some space to write to a file. Or, in case a file is not found, you can give a "file not found" error to the user, who can then retry the operation using a different file name, for example. Exceptions are something you don't usually define in your application, but the system raises them in *exceptional situations*, like when a programming error is encountered.

An exception can be raised, for example, when memory is low, and memory allocation cannot be performed, or when a programming error results in an array index out of bounds or null pointer. When an exception is thrown, the program cannot continue executing normally and might need to terminate. This is why many exceptions can be categorized as unrecoverable errors. In some cases, it is possible to recover from exceptions. Suppose a web service encounters a null pointer exception while handling an HTTP request. In that case, you can terminate the handling of the current request, return an error response to the client, and continue handling further requests normally. It depends on the software component how it should handle exceptional situations.

Do not confuse errors here with Java errors (inherited from the `Error` class). They are fatal errors that indicate a severe problem, a panic situation. As the Java documentation says, you should not catch these fatal errors in your code. Using the word "Error" in the class name to indicate a fatal situation is a wrong design decision made by the Java creators. A better name would have been a `FatalException`, for example.

Errors define situations where the execution of a function fails for some reason. Typical examples of errors are a file not found error, an error in sending an HTTP request to a remote service, or failing to parse a configuration file. Suppose a function can throw an error. Depending on the error, the function caller can decide how to handle the error. In case of transient errors, like a failing network request, the function caller can wait a while and call the function again. Or, the function caller can use a default value. For example, if a function tries to load a configuration file that does not exist, it can use some default configuration instead. And in some cases, the function caller cannot do anything but leave the error unhandled or catch the error but throw another error at a higher level of abstraction. Suppose a function tries to load a configuration file, but the loading fails, and no default configuration exists. In that case, the function cannot do anything but pass the error to its caller. Eventually, this error bubbles up in the call stack, and the whole process is terminated due to the inability to load the configuration. This is because the configuration is needed to run the application. Without configuration, the application cannot do anything but exit.

When defining error classes, define a base error class for your software component. For example, for the data exporter microservice, define a `DataExporterError` base error class. For each function that can throw, define a base error class at the same abstraction level as the function. That error class should extend the software component's base error class. For example, if you have a `parse(configStr)` function in the `ConfigParser` class, define a base error class for the function with the name `ConfigParseError`. If you have a `readFile` function, define a base error class with the name `FileReadError`. If you have a class where all methods can throw an error, it might be better to define a base error class at the class level. For example, if you have a `UserService` class with throwing methods, you can specify a `UserServiceError` class and throw an error of that class from the `UserService` class methods.

Below is an example of errors defined for the data exporter microservice:

```
public class DataExporterError extends RuntimeException {
  public DataExporterError(final String message) {
    super(message);
  }
}

public class FileReadError extends DataExporterError {
  public FileReadError(final String message) {
    super(message);
  }
}

public class ConfigParseError extends DataExporterError {
  public ConfigParseError(final String message) {
    super(message);
  }
}
```

Following the previous rules makes it easy to catch errors in the code because you can infer the error class name from the called method (or class) name. In the below example, we can infer the `FileReadError` error class name from the `readFile` method name:

```
try {
  final String fileContents = fileReader.readFile(...);
} catch (final FileReadError error) {
  // Handle error
}
```

You can also catch all user-defined errors using the software component's base error class in the catch clause. The below two examples have the same effect.

```
try {
  final String configFileContents = fileReader.readFile(...);
  return configParser.parse(configFileContents);
} catch (final FileReadError | ConfigParseError error) {
  // Handle error situation
}
```

```
try {
  final String configFileContents = fileReader.readFile(...);
  return configParser.parse(configFileContents);
} catch (final DataExporterError error) {
  // Handle error situation
}
```

Don't catch the language's base exception class or some other too-generic exception class because that will catch, in addition to all user-defined errors, exceptions, like null pointer exceptions, which is probably not what you want. So, do not catch a too-generic exception class like this:

```
try {
  final String configFileContents = fileReader.readFile(...);
  return configParser.parse(configFileContents);
} catch (final Exception exception) {
  // Do not use! Catches all exceptions
}
```

Also, do not catch the `Throwable` class in Java because it will also catch any fatal errors that are not meant to be caught:

```
try {
  final String configFileContents = fileReader.readFile(...);
  return configParser.parse(configFileContents);
} catch (final Throwable throwable) {
  // Do not use! Catches everything including
  // all exceptions and fatal errors
}
```

Catch all exceptions only in special places in your code, like in the main function or the main loop, like the loop in a web service processing HTTP requests or the main loop of a thread. Below is an example of correctly catching the language's base exception class in the main function. When you catch an unrecoverable exception in the main function, log it and exit the process with an appropriate error code. When you catch an unrecoverable error in a main loop, log it and continue the loop if possible.

```
public static void main(final String[] args) {
  // ...

   try {
     dataExporter.run(...);
   } catch (final Exception exception) {
     logger.log(exception);
     System.exit(1);
   }
}
```

You can also make your software component throw exceptions if needed. In the below example, we have created a base exception class for the data exporter microservice and derived one specific exception from it:

```
public class DataExporterException extends RuntimeException {
  public DataExporterException(final String message) {
    super(message);
  }
}

public class MySpecificException extends DataExporterException {
  public MySpecificException(final String message) {
    super(message);
  }
}
```

Using the above-described rules, you can make your code future-proof or forward-compatible so that adding new errors to be thrown from a function in the future is possible. Let's say that you are using a `fetchConfig` function like this:

```
try {
  final var configuration = configFetcher.fetchConfig(configUrl);
} catch (final ConfigFetchError error) {
  // Handle error
}
```

Your code should still work if a new type of error is thrown from the `fetchConfig` function. Let's say that the following new errors could be thrown from the `fetchConfig` function:

- Malformed URL error
- Server not found error
- Connection timeout error

When classes for these new errors are implemented, they must extend the function's base error class, in this case, the `ConfigFetchError` class. Below are the new error classes defined:

```
public class MalformedConfigUrlError extends ConfigFetchError {
  public MalformedConfigUrlError(final String message) {
    super(message);
  }
}
```

```
public class ConfigServerNotFoundError extends ConfigFetchError {
  public ConfigServerNotFoundError(final String message) {
    super(message);
  }
}

public class ConfigFetchTimeoutError extends ConfigFetchError {
  public ConfigFetchTimeoutError(final String message) {
    super(message);
  }
}
```

You can later enhance your code and handle different errors thrown from the `fetchConfig` differently. For example, you might want to handle a `ConfigFetchTimeoutError` so that the function will wait a while and then retry the operation because the error can be transient:

```
try {
  final var configuration = configFetcher.fetchConfig(configUrl);
} catch (final ConfigFetchTimeoutError error) {
  // Retry after a while
} catch (final MalformedConfigUrlError error) {
  // Inform caller that URL should be checked
} catch (final ConfigServerNotFoundError error) {
  // Inform caller that URL host/port cannot be reached
} catch (final ConfigFetchError error) {
  // Handle possible other error situations
  // This will catch any new exception that could be thrown
  // from the 'fetchConfig' function in the future
}
```

In the above examples, we handled thrown errors correctly, but you can easily forget to handle a thrown error. This is because nothing in the function signature tells you whether the function can throw or not. The only way to find out is to check the documentation (if available) or investigate the source code (if available). This is one of the biggest problems regarding error handling because you must know and remember that a function can throw, and you must remember to catch and handle errors. You don't always want to handle an error immediately, but still, you must be aware that the error will bubble up in the call stack and should be dealt with eventually somewhere in the code.

The solution to this problem is to make throwing errors more explicit:

> *Use a* `try` *prefix in the function name if the function can throw an error.*

This is a straightforward rule. If a function can throw an error, name the function so that its name starts with `try`. This makes it clear to every caller that the function can throw an error, and the caller should be prepared for that. For the caller of the function, there are three alternatives to deal with a thrown error:

1. Catch the base error class of the called function (or class or software component) and handle the error, e.g., catch `DataFetchError` if you are calling a function named `tryFetchData`.
2. Catch the base error class of the called function (or class or software component) and throw a

new error on a higher level of abstraction. Now you also have to name the calling function with a `try` prefix.

3. Don't catch errors. Let them propagate upwards in the call stack. Now you also have to name the calling function with a `try` prefix.

Here is an example of alternative 1:

```
public class ConfigFetcher {
  public Configuration fetchConfig(final String configUrl) {
    try {
      final var configDataStr = dataFetcher.tryFetchData(configUrl);
      return configParser.tryParse(configDataStr);
    } catch (final DataFetchError | ConfigParseError error) {
      // You could also catch DataFetchError and
      // ConfigParseError in different catch clauses
      // if their handling differs
      // You could also catch the base error class DataExporterError
      // of the software component
    }
  }
}
```

And here is an example of alternative 2:

```
public class ConfigFetcher {
  public Configuration tryFetchConfig(final String configUrl) {
    try {
      final var configDataStr = dataFetcher.tryFetchData(configUrl);
      return configParser.tryParse(configDataStr);
    } catch (final DataFetchError | ConfigParseError error) {
      // Error on higher level of abstraction is thrown
      // This function must be named with the 'try' prefix
      // to indicate that it can throw
      throw new ConfigFetchError(...);
    }
  }
}

public class DataExporter {
  public void initialize(...) {
    try {
      final var configuration = configFetcher.tryFetchConfig(...);
    } catch (final ConfigFetchError error) {
      // Handle error
    }
  }
}
```

And here is an example of alternative 3:

```
public class ConfigFetcher {
  public Configuration tryFetchConfig(final String configUrl) {
    final var configDataStr = dataFetcher.tryFetchData(configUrl);
    return configParser.tryParse(configDataStr);

    // No try-catch, all thrown errors from both tryFetchData
```

```
    // and tryParseConfiguration function calls propagate
    // to the caller and
    // this function must be named with the 'try' prefix
    // to indicate that it can throw
  }
}

public class DataExporter {
  public void initialize(...) {
    try {
      final var configuration = configFetcher.tryFetchConfig(...);
    } catch (final DataExporterError error) {
      // In this case you must catch the base error class of
      // the software component (DataExporterError), because
      // you don't know what errors tryFetchConfig can
      // throw, because no ConfigFetchError class
      // has been defined
    }
  }
}
```

As a side note, a linting rule that enforces the correct naming of throwing functions could be developed. The rule should force the function name to have a `try` prefix if the function throws or propagates errors. A function propagates errors when it calls a throwing (try-prefixed) method outside a try-catch block.

You can also create a library that has try-prefixed functions that wrap throwing functions that don't follow the try-prefix rule:

JsonParser.js

```
export default class JsonParser {
  static tryParse(json, reviver) {
    return JSON.parse(json, reviver);
  }
}
```

When using a web framework, the framework usually provides an error-handling mechanism. The framework catches an error and maps it to an HTTP response with an HTTP status code indicating a failure. Typically the default status code is 500 *Internal Server Error*. For example, when using Spring Boot, you can mark your custom error classes with a `@ResponseStatus` annotation:

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
class MyError extends RuntimeException {
  // ...
}
```

There are also alternative ways to map errors to HTTP responses in the Spring framework.

When you utilize a web framework's error-handling mechanism, there is no benefit in naming throwing functions with the try-prefix. You can opt out of the try-prefix rule.

It is usually a good practice to document the used error handling mechanism in the software

component documentation.

Handling Checked Exceptions in Java

You can use checked exceptions in Java when defining your software component's errors. Using checked exceptions helps you to remember to handle errors or let them propagate upwards in the call stack.

When using Java's checked exceptions, define the software component's base error class to extend the `Exception` class instead of the `RuntimeException` class. When a function throws a checked exception, it is unnecessary to prefix the function name with the `try` prefix. Below is an example of defining checked exceptions:

```
public class DataExporterError extends Exception {
  public DataExporterError(final String message) {
    super(message);
  }
}

public class DataFetchError extends DataExporterError {
  public DataFetchError(final String message) {
    super(message);
  }
}

public class ConfigParseError extends DataExporterError {
  public ConfigParseError(final String message) {
    super(message);
  }
}

public class InitializationError extends DataExporterError {
  public InitializationError(final String message) {
    super(message);
  }
}

public class DataFetcher {
  public String fetchData(...)
  throws DataFetchError {
    // ...
  }
}

public class ConfigParser {
  public Configuration parse(...)
  throws ConfigParseError {
    // ...
  }
}

public class DataExporter {
  public void initialize(...) throws InitializationError {
    try {
      final var configDataStr = dataFetcher.fetchData(configUrl);
```

```
      final var configuration = configParser.parse(configDataStr);

      // ...
    } catch (final DataFetchError | ConfigParseError error) {
      throw new InitializationError(error);
    }
  }
}
```

Later, it is possible to modify the implementation of the `parse` function to throw other errors that derive from the `ConfigParseError` class. This kind of change does not require modifications to other parts of the codebase.

On higher levels of the software component code, you can also use the base error class of the software component in the `throws` clause to propagate errors upwards in the call stack:

```
public class DataExporter {
  public void initialize(...) throws DataExporterError {
    final var configDataStr = dataFetcher.fetchData(configUrl);
    final var configuration = configParser.parse(configDataStr);

    // ...
  }
}
```

Returning Errors

As an alternative to throwing errors, it is possible to communicate erroneous behavior to the function caller using a return value. Using an exception-handling mechanism provides some advantages over returning errors. When a function can return an error, you must always check for the error right after the function call. This can cause the code to contain nested if-statements, which hinders code readability. The exception-handling mechanism allows you to propagate an error to a higher level in the call stack. You can also execute multiple function calls that can fail inside a single `try` block and provide a single error handler in the `catch` block. Some languages do not provide an exception-handling mechanism meaning you must return errors from functions. In languages like C++, you can optimize mission-critical code by returning error values or indicators instead of throwing exceptions.

Returning Failure Indicator

You can return a failure indicator from a failable function when the function does not need to return any additional value. It is enough to return a failure indicator from the function when there is no need to return any specific error code or message. This can be because there is only one reason the function can fail, or function callers are not interested in error details. To return a failure indicator, return a boolean value from the function: `true` means a successful operation, and `false` indicates a failure:

```
bool performTask(...)
{
  bool taskWasPerformed;

  // Perform the task and set the value of 'taskWasPerformed'

  return taskWasPerformed;
}
```

Returning an Optional Value

Suppose a function should return a value, but the function call can fail, and there is precisely one cause why the function call can fail. In this case, return an optional value from the function. In the below example, getting a value from the cache can only fail when no value for a specific key is stored in the cache. We don't need to return any error code or message.

Cache.java

```
public interface Cache<K, V> {
  void add(K key, V value);
  Optional<V> get(K key);
}
```

Returning an Error Object

When you need to provide details about an error to a function caller, you can return an error object from the function:

BackendError.ts

```
export type BackendError = {
  statusCode: number;
  errorCode: number;
  message: string;
};
```

If a function does not return any value but can produce an error, you can return either an error object or `null` in languages that have `null` defined as a distinct type and the language supports type unions (e.g., TypeScript):

DataStore.ts

```
export interface DataStore {
  updateEntity<T extends Entity>(...):
    Promise<BackendError | null>;
}
```

Alternatively, return an optional error. Below is an example in Java:

```
import lombok.experimental.Value;

@Value
public class BackendError {
```

```
  int statusCode;
  int errorCode;
  String message;
}

public interface DataStore {
  <T extends Entity> Optional<BackendError> updateEntity(...);
}
```

Suppose a function needs to return a value or an error. In that case, you can use a 2-tuple (i.e., a pair) type, where the first value in the tuple is the actual value or `null` in case of an error and the second value in the tuple is an error object or `null` value in case of a successful operation. Below are examples in TypeScript and Java. In the Java example, you, of course, need to return optionals instead of nulls.

DataStore.ts

```
export class DataStore {
  createEntity<T extends Entity>(...):
    Promise<[T, null] | [null, BackendError]>;
}
```

DataStore.java

```
import org.javatuples.Pair;

public interface DataStore {
  <T extends Entity> Pair<Optional<T>, Optional<BackendError>>
  createEntity(...);
}
```

The above Java example is cumbersome to use, and the type definition looks long. We should use an `Either` type here, but Java does not have that. Either type contains one of two values, either a left value or a right value. The left value is the value returned by the function when the operation is successful, and the right value is an error. The `Either` type can be defined as follows:

Either.java

```
public class Either<L, R>
{
  private final Optional<L> maybeLeftValue;
  private final Optional<R> maybeRightValue;

  private Either(
    final Optional<L> maybeLeftValue,
    final Optional<R> maybeRightValue
  ) {
    this.maybeLeftValue = maybeLeftValue;
    this.maybeRightValue = maybeRightValue;
  }

  public static <L, R> Either<L, R> withLeft(
    final L value
  ) {
    return new Either<>(Optional.of(value), Optional.empty());
  }
```

```
  public static <L, R> Either<L, R> withRight(
    final R value
  ) {
    return new Either<>(Optional.empty(), Optional.of(value));
  }

  public boolean hasLeftValue() {
    return maybeLeftValue.isPresent();
  }

  public <T> Either<T, R> mapLeft(
    Function<? super L, ? extends T> mapper
  ) {
    return new Either<>(maybeLeftValue.map(mapper),
                        maybeRightValue);
  }

  public <T> Either<L, T> mapRight(
    Function<? super R, ? extends T> mapper)
  {
      return new Either<>(maybeLeftValue,
                          maybeRightValue.map(mapper));
  }

  public <T> T map(
    Function<? super L, ? extends T> leftValueMapper,
    Function<? super R, ? extends T> rightValueMapper)
  {
    return maybeLeftValue.<T>map(leftValueMapper)
      .orElseGet(() ->
        maybeRightValue.map(rightValueMapper).get());
  }

  public void apply(
    Consumer<? super L> leftValueConsumer,
    Consumer<? super R> rightValueConsumer
  ) {
    maybeLeftValue.ifPresent(leftValueConsumer);
    maybeRightValue.ifPresent(rightValueConsumer);
  }
}
```

Now we can use the new `Either` type and rewrite the example as follows:

DataStore.java

```
public interface DataStore {
  <T extends Entity> Either<T, BackendError> createEntity(...);
}
```

Adapt to Wanted Error Handling Mechanism

You can adapt to a desired error-handling mechanism by creating an adapter method. For example, if a library has a throwing method, you can create an adapter method returning an optional value or error object. Below is a `tryCreate` factory method in a `VInt` class that can throw:

VInt.ts

```
class VInt {
```

```
  // ...

  private constructor(...) {
    // ...
  }

  // this will throw if invalid 'value' is given
  // that doesn't match the 'validationSpec'
  static tryCreate<VSpec extends string>(
    validationSpec: IntValidationSpec<VSpec>,
    value: number
  ): VInt<VSpec> | never {
    // constructor can throw
    return new VInt(validationSpec, value);
  }

  // ...
}
```

We can create a `VIntFactory` class with an adapter method for the `tryCreate` factory method in the `VInt` class. The `VIntFactory` class offers a non-throwing `create` method:

VIntFactory.ts

```
class VIntFactory {
  static create<VSpec extends string>(
    validationSpec: IntValidationSpec<VSpec>,
    value: number
  ): VInt<VSpec> | null {
    try {
      return VInt.tryCreate(validationSpec, value);
    } catch {
      return null;
    }
  }
}
```

We can also create a method that does not throw but returns either a value or an error:

VIntFactory.ts

```
class VIntFactory {
  static createOrError<VSpec extends string>(
    validationSpec: IntValidationSpec<VSpec>,
    value: number
  ): [VInt<VSpec>, null] | [null, Error] {
    try {
      return [VInt.tryCreate(validationSpec, value), null];
    } catch (error) {
      return [null, error as Error];
    }
  }
}
```

We can also introduce a simplified version of the `Either` type for TypeScript:

Either.ts

```
export type Either<L, R> = [L, null] | [null, R];
```

Now we can rewrite the above example like this:

VIntFactory.ts

```
class VIntFactory {
  static createOrError<VSpec extends string>(
    validationSpec: IntValidationSpec<VSpec>,
    value: number
  ): Either<VInt<VSpec>, Error> {
    try {
      return [VInt.tryCreate(validationSpec, value), null];
    } catch (error) {
      return [null, error as Error];
    }
  }
}
```

Asynchronous Function Error Handling

Asynchronous functions are functions that usually can fail. They often execute I/O operations like file or network I/O. For a failable asynchronous operation, you must remember to handle the failure case. For this reason, it is suggested to name a failable asynchronous operation using the same `try` prefix used in function names that can throw. Below are two examples of handling an asynchronous operation failure in JavaScript/TypeScript:

```
tryMakeHttpRequest(url).then((value) => {
  // success
}, (error) => {
  // Handle error
});
```

```
tryMakeHttpRequest(url).then((value) => {
  // success
}).error((error) => {
  // Handle error
});
```

As you can see from the above examples, it is easy to forget to add the error handling. It would be better if there was a `thenOrCatch` method in the `Promise` class that accepted the following kind of callback:

```
tryMakeHttpRequest(url).thenOrCatch(([value, error]) => {
  // Now it is harder to forget to handle an error
  // Check 'error' before using the 'value'
});
```

You can make asynchronous function calls synchronous. In JavaScript/TypeScript, this can be done using the `async` and `await` keywords. A failable asynchronous operation made synchronous can throw. Below is the same example as above made synchronous:

```
async function fetchData() {
```

```
  try {
    await tryMakeHttpRequest(url);
  } catch {
    // Handle error
  }
}
```

Functional Exception Handling

The below `Failable<T>` class can be used in functional error handling. A `Failable<T>` object represents either a value of type `T` or an instance of the `RuntimeException` class.

Failable.java

```
public class Failable<T> {
  private final Either<T, RuntimeException> valueOrError;

  private Failable(
    final Either<T, RuntimeException> valueOrError
  ) {
    this.valueOrError = valueOrError;
  }

  public static <T> Failable<T> withValue(
    final T value
  ) {
    return new Failable<>(Either.withLeft(value));
  }

  public static <T> Failable<T> withError(
    final RuntimeException error
  ) {
    return new Failable<>(Either.withRight(error));
  }

  public T orThrow(
    final Class<? extends RuntimeException> ErrorClass
  ) {
    return valueOrError.map(
      (value) -> value,
      (error) -> {
          try {
            throw (RuntimeException)ErrorClass
              .getConstructor(String.class)
              .newInstance(error.getMessage());
          } catch (InvocationTargetException |
                   InstantiationException |
                   IllegalAccessException |
                   IllegalArgumentException |
                   NoSuchMethodException exception) {
            throw new RuntimeException(exception);
          }
      });
  }

  public T orElse(final T otherValue) {
    return valueOrError.map(value -> value,
                            error -> otherValue);
  }
```

```
  public <U> Failable<U> mapValue(
    final Function<? super T, ? extends U> mapper
  ) {
    return new Failable<>(valueOrError.mapLeft(mapper));
  }

  public Failable<T> mapError(
    final Function<? super RuntimeException,
                   ? extends RuntimeException> mapper
  ) {
    if (valueOrError.hasLeftValue()) {
      final var error =
        new RuntimeException(mapper
          .apply(new RuntimeException(""))
          .getMessage());

      return Failable.withError(error);
    } else {
      return new Failable<>(valueOrError.mapRight(mapper));
    }
  }
}
```

In the below example, the `readConfig` method returns a `Failable<Configuration>`. The `tryInitialize` function either obtains an instance of `Configuration` or throws an error of type `InitializationError`.

```
public void tryInitialize() {
  final var configuration = configReader
    .readConfig(...)
    .orThrow(InitializationError.class);
}
```

The benefit of the above functional approach is that it is shorter than an entire try-catch block. The above functional approach is also as understandable as a try-catch block. Remember that you should write the shortest, most understandable code. When a method returns a failable, you don't have to name the method with the `try` prefix because the method does not throw.

You can also use other methods of the `Failable` class. For example, a default value can be returned with the `orElse` method:

```
public void initialize() {
  final var configuration = configReader
    .readConfig(...)
    .orElse(new DefaultConfiguration());
}
```

You can also transform multiple imperative failable statements into functional failable statements. For example, instead of writing:

```
public void tryInitialize() {
  try {
    final var configDataStr = dataFetcher.tryFetchData(configUrl);
    final var configuration = configParser.tryParse(configDataStr);
```

```
  } catch (final DataExporterError error) {
    throw new InitializationError(error.getMessage());
  }
}
```

You can write:

```
public void tryInitialize() {
  try {
    final var configuration = dataFetcher
      .fetchData(configUrl)
      .mapValue(configParser::parse)
      .orThrow(InitializationError.class);
}
```

It is error-prone to use failable imperative code together with functional programming constructs. Let's assume we have the below TypeScript code that reads and parses multiple configuration files to a single configuration object:

```
configFilePathNames
  .reduce((accumulatedConfig, configFilePathName) => {
    const configJson = fs.readFileSync(configFilePathName, 'utf-8');
    const configuration = JSON.parse(configJson);
    return { ...accumulatedConfig, ...configuration };
}, {});
```

In the above example, it is easy to forget to handle errors because the throwability of the `reduce` function depends on the supplied callback function. We cannot use the try-prefix anywhere in the above example. What we can do is the following:

```
function tryReadConfig(
  accumulatedConfig: Record<string, unknown>,
  configFilePathName: string
) {
  const configJson = fs.readFileSync(configFilePathName, 'utf-8');
  const configuration = JSON.parse(configJson);
  return { ...accumulatedConfig, ...configuration };
}

export function getConfig(
  configFilePathNames: string[]
): Record<string, unknown> {
  try {
    return configFilePathNames.reduce(tryReadConfig, {});
  } catch (error) {
    // ...
  }
}
```

We have now added the `try` prefix, but the code could read better. A better alternative is to use a functional programming construct, `Failable<T>`, to return a failable configuration. The `Failable<T>` class implementation in TypeScript is not presented here, but it can be implemented similarly to Java. Below is an example of using the `Failable<T>` class:

```
function accumulatedConfigOrError(
  accumulatedConfigOrError: Failable<Record<string, unknown>>,
  configFilePathName: string
): Failable<Record<string, unknown>> {
  try {
    const configJson = fs.readFileSync(configFilePathName, 'utf-8');
    const config = JSON.parse(configJson);

    return accumulatedConfigOrError.mapValue(accumulatedConfig =>
      ({ ...accumulatedConfig, ...config }));
  } catch (error: any) {
    return accumulatedConfigOrError.mapError(accumulatedError =>
      new Error(`${accumulatedError.message}\n${error.message}`)
    );
  }
}

export function getConfig(
  configFilePathNames: string[]
): Failable<Record<string, unknown>> {
  return configFilePathNames.reduce(
    accumulatedConfigOrError,
    Failable.withValue({})
  );
}
```

Stream Error Handling

Handling errors for a stream is also something that can be easily forgotten. Streams are usually used for I/O operations that can fail. You should be prepared for error handling when using a stream. In JavaScript/TypeScript, an error handler for a stream can be registered using the stream's `on` method in the following way: `stream.on('error', () => { ... })`.

Below is an example of using a stream:

```
// ...

const writeStream = fs.createWriteStream(filePathName);

this.writeStream.on('error', (error) => {
   // Handle errors
});

writeStream.write(...);
// More writes...

writeStream.close();
```

How could we improve developer experience with streams, so that error handling is not forgotten? One solution is to add an error handler callback parameter to the stream factory method. This callback will be called upon an error. If no error handling is needed, a `null` value could be given for the callback. This way, a developer creating a stream can't forget to supply an error handler function.

Don't Pass or Return Null Principle

This principle is for languages like Java and C++ that don't implement null values as separate types. TypeScript implements nulls as distinct types when the `strictNullChecks` configuration parameter is set to true, and you should always set it to true. So, this principle does not apply to TypeScript.

The null value is the misguided invention of British computer scientist *Tony Hoare* who coined his invention of null references as a "billion-dollar mistake". The reason is quite evident because we all have done it: forgetting to handle a null value. And when we don't handle a null value, we pass it to other functions that never expect to be called with a null value. Eventually, this will lead to a null value exception thrown somewhere in the code.

When you return a value from a function, never return a null value. You should return an optional value instead. In the below example, we are returning an optional value for a key in a map because there can be no value associated with a particular key in the map.

```
// BAD!
public class Map<K, V> {
  public V get(final K key) {
    if (...) {
      // ...
    } else {
      return null;
    }
  }
}


// GOOD!
public class Map<K, V> {
  public Optional<V> get(final K key) {
    // ...
  }
}
```

When you pass arguments to a function, never pass a null value. The called function usually never expects to be called with null arguments. Suppose a function expects an argument that can be missing. In that case, the function can define a default value for that argument (possible in C++ and JavaScript/TypeScript, but not in Java), or an overloaded function can be defined where the optional argument is not present. A function can also be defined so that an argument has an optional type, but you should prefer an optional parameter or an overloaded version of the function.

Avoid Off-By-One Errors Principle

Off-by-one errors usually result from the fact that collections in programming languages are indexed with zero-based indexes. Zero-based indexing is unnatural for human beings but excellent for computers. However, programming languages should be designed with humans in mind. People never speak about getting the zeroth value of an array. We speak of getting the first value in the array. As the null value was called a billion-dollar mistake, I would call the zero-based indexing another billion-dollar mistake. Let's hope that someday we get a programming language with one-based indexing! But then we must unlearn the zero-based indexing habit...and that's another problem.

Below are two examples of programming errors in JavaScript that are easy to make if you are not careful enough:

```
for (let index = 0; index <= array.length; index++) {
  // ...
}

for (let index = 0; index < array.length - 1; index++) {
  // ...
}
```

In the first example, there should be '<' instead of '<=', and in the latter example, there should be '<=' instead of '<'. Fortunately, the above mistakes can be avoided using modern programming language constructs like Java's enhanced for-loop or C++'s range-based for-loop or functional programming.

Below are two examples of avoiding off-by-one errors in Java:

```
for (final var value : values) {
  // ...
}

values.stream().forEach(value -> ...);
```

Some languages, like JavaScript, offer a nice way to access an array's last element(s). Instead of writing `array[array.length - 1]`, you can write `array.at(-1)`. And similarly, `array[array.length - 2]` is the same as `array.at(-2)`. You can think that a negative index is a one-based index starting from the end of an array.

Let's consider the description of JavaScript's `slice` method:

> *The slice() method returns a shallow copy of a portion of an array into a new array object selected from start index to end index (end not included).*

The problem here is the 'end not included' part. Many people, by default, think that if given a range, it is inclusive, but in the case of the `slice` method, it is inclusive at the beginning and exclusive at

the end: `[start, end[`. This kind of function definition that is against first assumptions can easily cause off-by-one errors. It would be better if the `slice` method by default works with an inclusive range `[start, end]`.

Additionally, unit tests are your friend when trying to spot off-by-one errors. So remember to write unit tests for the edge cases, too.

Be Critical When Googling Principle

> *You should always analyze code taken from the web to see if it meets the criteria for production code.*

We all have done it, and we have done it hundreds of times: googled for answers. Usually, you find good resources by googling, but the problem often is that examples in the googled results are not necessarily production quality. One specific thing missing in them is error handling. If you copy and paste code from a website, it is possible that errors are not handled appropriately. You should always analyze the copy-pasted code to see if error handling needs to be added.

When you provide answers for other people, try to make the code as production-like as possible. In Stack Overflow, you find the most up-voted answer right below the question. If the answer is missing error handling, you can comment on that and let the author improve their answer. You can also up-vote an answer that seems the most production ready. Usually, the most up-voted answers are pretty old. For this reason, it is useful to scroll down to see if a more modern solution fits your need better. And you can also up-vote that more modern solution so it will become ranked higher in the list of answers.

Regarding open source libraries, the first examples in their documentation can describe only the "happy path" usage scenario, and error handling is described only in later parts of the documentation. This can cause problems if you copy-paste code from the "happy path" example and forget to add error handling. For this reason, open-source library authors should give production-quality examples early in the documentation.

Optimization Principle

> *Avoid premature optimization. Premature optimization may hinder crafting a proper object-oriented design for a software component. Measure unoptimized performance first. Then decide if optimization is needed. Implement optimizations one by one and measure the performance after each optimization round to determine if the optimization matters. You can then utilize gained knowledge in future projects only to make optimizations that give a*

significant enough performance boost. Sometimes you can make performance optimization in an early phase of a project if you know that a particular optimization is needed (e.g., from previous experience) and the optimization can be implemented without negatively affecting the object-oriented design.

Optimization Patterns

The following optimization patterns are described in this section:

- Optimize busy loops only pattern
- Remove unnecessary functionality pattern
- Copy memory in chunks pattern (C++)
- Object pool pattern
- Replace virtual methods with non-virtual methods pattern (C++)
- Inline methods pattern (C++)
- Use unique pointer pattern (C++)
- Share identical objects a.k.a flyweight pattern

Optimize Busy Loops Only Pattern

Optimizations should primarily target only the busy loop or loops in a software component. Busy loops are the loops in threads that execute over and over again, possibly thousands or more iterations in a second. Performance optimization should not target functionality that executes only once or a couple of times during the software component's lifetime, and running that functionality does not take a long time. For example, an application can have configuration reading and parsing functionality when it starts. This functionality takes a short time to execute. It is not reasonable to optimize that functionality because it runs only once. It does not matter if you can read and parse the configuration in 200 or 300 milliseconds, even if there is a 50% difference in performance.

Let's use the data exporter microservice as an example. Our data exporter microservice consists of input, transformer, and output parts. The input part reads messages from a data source. We cannot affect the message reading part if we use a 3rd party library for that purpose. Of course, if multiple 3rd party libraries are available, it is possible to craft performance tests and evaluate which 3rd party library offers the best performance. If there are several 3rd party libraries available for the same functionality, we tend to use the most popular library or a library we know beforehand. If performance is an issue, we should evaluate different libraries and compare their performances.

The data exporter microservice has the following functionality in its busy loop: decode an input message to an internal message, perform transformations, and encode an output message. Decoding an input message requires decoding each field in the message. Let's say there are 5000 messages handled per second, and each message has 100 fields. During one second, 50000 fields

must be decoded. This reveals that the optimization of the decoding functionality is crucial. The same applies to output message encoding. We at Nokia have implemented the decoding and encoding Avro binary fields ourselves. We were able to make them faster than what was provided by a 3rd party library.

Remove Unnecessary Functionality Pattern

Removing unnecessary functionality is something that will boost performance. You should stop to think critically about your software component: Is my software component doing only the necessary things considering all circumstances?

Let's consider the data exporter's functionality. It is currently decoding an input message to an internal message. This internal message is used when making various transformations to the data. Transformed data is encoded to a wanted output format. The contents of the final output message can be a small subset of the original input message. This means that only a tiny part of the decoded message is used. In that case, it is unnecessary to decode all the fields of an input message if, for example, only 10% of the fields are used in the transformations and output messages. By removing unnecessary decoding, we can improve the performance of the data exporter microservice.

Copy Memory in Chunks Pattern (C++)

If you have a contiguous memory chunk, copy it using `memcpy`. Don't copy memory byte by byte in a for-loop. The implementation of the `memcpy` function is optimized by a C++ compiler to produce machine code that optimally copies various sizes of memory chunks. Instead of copying a memory chunk byte by byte, it can, for example, copy memory as 64-bit values in a 64-bit operating system.

In the data exporter microservice, there is the possibility that the input message format and output message format are the same, e.g., Avro binary. We can have a situation where an Avro record field can be copied as such from an input message to an output message without any transformation. In that case, decoding that record field is unnecessary functionality, and we can skip that. What we will do instead is copy a chunk of memory. An Avro record field can be relatively large, even 200 bytes consisting of 40 subfields. We can now skip the decoding and encoding of those 40 subfields. We simply copy 200 bytes from the input message to the output message.

Object Pool Pattern

In garbage-collected languages like JavaScript and Java, the benefit of using an object pool is clear from the garbage-collection point of view. In the object pool pattern, objects are created only once and then reused. This will take pressure away from garbage collection. If we didn't use an object pool, new objects could be created in a busy loop repeatedly, and soon after they were created, they could be discarded. This would cause many objects to be made available for garbage collection in a short period of time. Garbage collection takes processor time, and if the garbage collector has a lot

of garbage to collect, it can slow the application down for an unknown duration at unknown intervals.

Replace Virtual Methods with Non-Virtual Methods Pattern (C++)

If you are using a lot of calls to virtual functions in a busy loop, there will be some overhead in checking which virtual method to call due to dynamic dispatch. In C++, this is done using virtual tables (*vtables*) which are used to check which actual method will be called. The additional *vtable* check can negatively affect performance in busy loops if virtual methods are called frequently. For example, the data exporter microservice's busy can call an Avro binary decoding and encoding function 50000 times a second. We could optimize these calls by implementing Avro binary decoding functions as non-virtual (if previously declared as virtual functions). Non-virtual functions don't need to check the *vtable*, so the call to the function is direct.

Inline Methods Pattern (C++)

Suppose you have made the optimization of making a virtual method non-virtual. One more optimization could still be made if the method is small: inline the method. Inlining a method means that calls to the method are eliminated, and the code of the method is placed at the sites where the calls to the method are made. So, the method does not need to be called at all when it has been inlined. In the data exporter microservice, we made the Avro binary encoding and decoding functions non-virtual, and now we can make them also inlined to speed up the microservice. However, a C++ compiler can decide whether an inline function is really inlined or not. We cannot be 100% sure if the function is inlined. It's up to the compiler. When we define a function as an inline function with the C++'s `inline` keyword, we are just giving a hint to the compiler that the function should be inlined. Only non-virtual methods can be inlined. Virtual methods cannot be inlined because they require checking the *vtable* to decide which method should be called.

Use Unique Pointer Pattern (C++)

If you are using shared pointers, they need to keep the reference count to the shared pointer up to date. In a busy loop, if you use a shared pointer, say a hundred thousand times a second, it starts to show a difference whether you use a shared pointer or a unique pointer (`std::unique_ptr`). A unique pointer has little to no overhead compared to a raw pointer. For this reason, there is no need to use a raw pointer in modern C++. It would not bring much to the table performance-wise. And if you use a raw pointer, you must remember to release the allocated memory associated with the raw pointer by yourself. If you don't need a pointed object to be shared by multiple other objects, you can optimize your code in busy loops by changing shared pointers to unique pointers. Unique pointers always have only one owner, and multiple objects cannot share them.

Share Identical Objects a.k.a Flyweight Pattern

If your application has many objects with some identical properties, those parts of the objects with identical properties are wasting memory. You should extract the common properties to a new class and make the original objects reference a shared object of that new class. Now your objects share a single common object, and possibly significantly less memory is consumed. This design pattern is called the *flyweight pattern* and was described in more detail in the earlier chapter.

Testing Principles

Testing is traditionally divided into two categories: functional and non-functional testing. This chapter will first describe the functional testing principles and then the non-functional testing principles.

Functional Testing Principles

Functional testing is divided into three phases:

- Unit testing
- Integration testing
- End-to-end (E2E) testing

Functional test phases can be described with the *testing pyramid*:

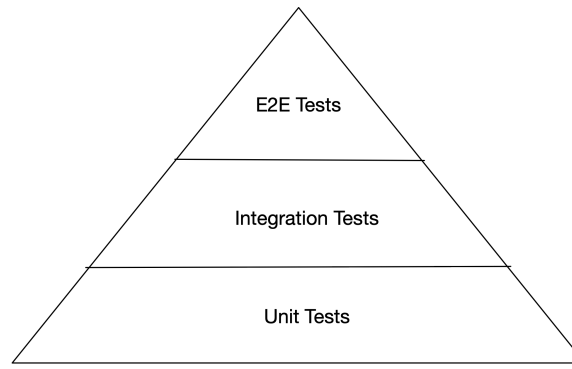

Figure 5.1 Testing Pyramid

The testing pyramid depicts the relative number of tests in each phase. Most tests are unit tests. The second most tests are integration tests, and the fewest are E2E tests. Unit tests should cover the whole codebase of a software component. Unit testing focuses on testing individual public functions as units (of code). Software component integration tests cover the integration of the unit-tested functions to a complete working software component, including testing the interfaces to external services. Examples of external services are a database, a message broker, and other microservices. E2E testing focuses on testing the end-to-end functionality of a complete software system.

Unit Testing Principle

> *Unit tests should test the functionality of public functions as isolated units with as high coverage as possible. The isolation means that dependencies (other classes/modules/services) are mocked.*

Unit tests should be written for public functions only. Do not try to test private functions separately. They should be tested indirectly when testing public functions. Below is a JavaScript example:

parseConfig.js

```
import { doSomething } from 'other-module';

function readFile(...) {
   // ...
}

export default function parseConfig(...) {
   // ...
   // readFile(...)
   // doSomething(...)
   // ...
}
```

In the above module, there is one public function, `parseConfig`, and one private function, `readFile`. In unit testing, you should test the public `parseConfig` function in isolation and mock the `doSomething` function, which is imported from another module. And you indirectly test the private `readFile` function when testing the public `parseConfig` function.

Below is the above example written in Java. You test the Java version in a similar way as the JavaScript version. You write unit tests for the public `parseConfig` method only. Those tests will test the private `readFile` function indirectly. You must supply a mock instance of the `OtherClass` class for the `ConfigParser` constructor.

```
public class OtherClass {
   // ...

   public void doSomething(...) {
      // ...
   }
}

public class ConfigParser {
   private OtherClass otherClass;

   public ConfigParser(final OtherClass otherClass) {
      this.otherClass = otherClass
   }

   // ...

   public Configuration parseConfig(...) {
      // ...
      // readFile(...)
      // otherClass.doSomething(...)
      // ...
   }

   private String readFile(...) {
      // ...
   }
}
```

Unit tests should test all the functionality of a public function: happy path(s), possible failure situations, and edge cases so that each code line of the function is covered by at least one unit test.

Below are some examples of edge cases listed:

- Are the last loop counter value correct? This test should detect possible off-by-one errors
- Test with an empty array
- Test with the smallest allowed value
- Test with the biggest allowed value
- Test with a negative value
- Test with a zero value
- Test with a very long string
- Test with an empty string
- Test with floating-point values having different precisions
- Test with floating-point values that are rounded differently
- Test with a very small floating-point value
- Test with a very large floating-point value

Unit tests should not test the functionality of dependencies. That is something to be tested with integration tests. A unit test should test a function in isolation. If a function has one or more dependencies on other functions defined in different classes (or modules), those dependencies

should be mocked. A *mock* is something that mimics the behavior of a real object or function. Mocking will be described in more detail later in this section.

Testing functions in isolation has two benefits. It makes tests faster. This is a real benefit because you can have a lot of unit tests, and you run them often, so it is crucial that the execution time of the unit tests is as short as possible. Another benefit is that you don't need to set up external dependencies, like a database, a message broker, and other microservices, because you are mocking the functionality of the dependencies.

Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development process in which software requirements are formulated as test cases before the software is implemented. This is as opposed to the practice where software is implemented first, and test cases are written only after that.

I have been in the industry for almost 30 years, and when I began coding, there were no automated tests or test-driven development. Only starting from 2010 have I been writing automated unit tests. Due to this background, TDD has been quite difficult for me because there is something I have grown accustomed to: Implement the software first and then do the testing. If you have also learned it like that, switching to TDD can be quite hard.

The pure TDD cycle consists of the following steps:

1. Add a test for a specified functionality
2. Run all the tests (The just added test should fail because the functionality it is testing is not implemented yet)
3. Write the simplest possible code that makes the tests pass
4. Run all the tests. (They should pass now)
5. Refactor as needed (Existing tests should ensure that anything won't break)
6. Start again from the first step until all functionality is implemented, refactored, and tested

Let's continue with an example. Suppose there is the following user story in the backlog waiting to be implemented:

> *Parse configuration properties from a configuration string to a configuration object. Configuration properties can be accessed from the configuration object.*

Let's first write a test for the specified functionality:

ConfigParserTests.java

```
public class ConfigParserTests {
  private final ConfigParser configParser = new ConfigParserImpl();

  @Test
  public void testParse() {
    // GIVEN
    final var configStr = "propName1=value1\npropName2=value2";

    // WHEN
    final var configuration = configParser.parse(configStr);

    // THEN
    assertEquals(configuration.getPropertyValue("propName1"),
                 "value1");

    assertEquals(configuration.getPropertyValue("propName2"),
                 "value2");
  }
}
```

Now, if we run all the tests, we get a compilation error, which means that the test case we wrote won't pass yet. Next, we shall write the simplest possible code to make the test case both compile and pass:

```
public interface Configuration {
  String getPropertyValue(String propertyName);
}

public class ConfigurationImpl implements Configuration {
  private final Properties properties;

  public ConfigurationImpl(final Properties properties) {
    this.properties = properties;
  }

  public String getPropertyValue(final String propertyName) {
    return properties.getProperty(propertyName);
  }
}

public interface ConfigParser {
  Configuration parse(String configStr);
}

public class ConfigParserImpl implements ConfigParser {
  public Configuration parse(final String configStr) {
    final Properties properties = new Properties();

    // Load properties to 'properties' variable
    // from the 'configString'

    return new ConfigurationImpl(properties);
  }
}
```

We can now add new functionality. Let's say the `parse` function should throw an error if it cannot parse the configuration string. We can now repeat the TDD cycle from the beginning by creating a

failing test first:

```
public class ConfigParserTests {
  // ...

  @Test
  public void testParse_whenParsingFails() {
    // GIVEN
    final var configStr = "invalid";

    try {
      // WHEN
      configParser.tryParse(configStr);
      fail();
    } catch (final ConfigParseError error) {
      // THEN error was successfully thrown
    }
  }
}
```

Next, we should refactor the implementation to make the second test pass:

```
public interface ConfigParser {
  Configuration tryParse(String configStr);
}

public class ConfigParseError extends RuntimeException {
  public ConfigParseError(final String errorMessage,
                          final Throwable error) {
      super(errorMessage, error);
  }
}

public class ConfigParserImpl implements ConfigParser {
  public Configuration tryParse(final String configStr) {
    final Properties properties = new Properties();

    try {
      // Try load properties from the configStr

      return new ConfigurationImpl(properties);
    } catch (...) {
      throw new ConfigParseError(...);
    }
  }
}
```

We also need to refactor the first unit test to call `tryParse` instead of `parse`. We can continue adding test cases for additional functionality.

For me, the above-described TDD cycle sounds a bit cumbersome. But, there are clear benefits in creating tests beforehand. When tests are defined first, it is usually less likely that one forgets to test or implement something. This is because TDD better forces you to think about the function specification: happy path(s), edge and failure cases.

If you don't practice TDD and do the implementation always first, it is more likely you might forget

an edge case or a particular failure scenario. And when you haven't implemented it, you don't test it. You can have 100% unit test coverage for a function, but a particular edge case or failure scenario is left unimplemented and untested. This is what has happened to me, also.

As an alternative to the above-described TDD cycle, you can start function implementation by first considering the functionality the function should have. You can first think about the "happy path", which is the most common scenario for the function. Create an empty test case that contains only a statement that makes the test fail. You should put the `fail` call in the test case in order not to forget to implement the test case. Below is an example test case:

```
public class ConfigParserTests {
  // Tests the "happy path":
  // successful parsing of configuration
  @Test
  public void testParse() {
    fail();
  }
}
```

Next, think of all the other scenarios for the function: other happy paths, edge, and failure cases. And then create a failing test case for each of those scenarios, for example:

```
public class ConfigParserTests {
  @Test
  public void testParse_whenParsingFails() {
    fail();
  }

  @Test
  public void testParse_whenMandatoryPropIsMissing() {
    fail();
  }

  @Test
  public void testParse_whenOptionalPropIsMissing() {
    fail();
  }

  @Test
  public void testParse_whenPropHasInvalidName() {
    fail();
  }

  @Test
  public void testParse_whenPropHasInvalidType() {
    fail();
  }
}
```

Now you have a high-level specification of the function in the form of scenarios. Next, you can continue with the function implementation. After you have completed the function implementation, implement the test cases one by one, and remove the `fail` calls.

The benefit of this approach is that you don't have to switch continuously between the

implementation source code file and the test source code file. In each phase, you can focus on one thing:

1. Function specification
    1. Specify scenarios: What the function does, and what failures are possible? Are there edge cases?
    2. Implement scenarios as failing unit test cases
2. Function implementation
3. Implementation of unit tests

Naming Conventions

When functions to be tested are in a class, a respectively named class for unit tests should be created. For example, if there is a `ConfigParser` class, the respective class for unit tests should be `ConfigParserTests`. This way, it is easy to locate the file containing unit tests for a particular implementation class.

A test method name should start with a *test* prefix, after which the name of the tested method should come. For example, if the tested method is `parse`, the test method name should be `testParse`. There are usually several tests for a single function. All test method names should begin with *test<function-name>*, but the test method name should also contain a description of the scenario the test method tests: *test<function-name>_<scenario>*, for example: `testParse_whenParsingFails`.

When using the Jest testing library with JavaScript or TypeScript, unit tests are organized and named in the following manner:

```
describe('<class-name>', () => {
  describe('<public-method-name>', () => {
    it('should do this...', () => {
      // ...
    });

    it('should do other thing when...', () => {
      // ...
    });

    // Other scenarios...
  });
});

// Example:

describe('ConfigParser', () => {
  describe('parse', () => {
    it('should parse config string successfully', () => {
      // ...
    });

    it('should throw an error if parsing fails', () => {
```

```
      // ...
    });

    // Other scenarios...
  });
});
```

Mocking

Let's have a small Spring Boot example of mocking dependencies in unit tests. We have a service class that contains public functions for which we want to write unit tests:

SalesItemServiceImpl.java

```
@Service
public class SalesItemServiceImpl implements SalesItemService {
  @Autowired
  private SalesItemRepository salesItemRepository;

  @AutoWired
  private SalesItemFactory salesItemFactory;

  @Override
  public final SalesItem createSalesItem(
    final SalesItemArg salesItemArg
  ) {
    return salesItemRepository.save(
      salesItemFactory.createFrom(salesItemArg));
  }

  @Override
  public final Iterable<SalesItem> getSalesItems() {
    return salesItemRepository.findAll();
  }
}
```

In the Spring Boot project, we need to define the following dependency in the *build.gradle* file:

```
dependencies {
  // Other dependencies ...
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Now, we can create unit tests using JUnit, and we can use Mockito for mocking. Looking at the above code, we can notice that the `SalesItemServiceImpl` service depends on a `SalesItemRepository`. According to the unit testing principle, we should mock that dependency. Similarly, we should also mock the `SalesItemConverter` dependency:

SalesItemServiceTests.java

```
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.boot.test.context.SpringBootTest;
```

```
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.refEq;

@SpringBootTest
class SalesItemServiceTests {
  private static final String SALES_ITEMS_NOT_EQUAL =
    "Sales items not equal";

  private final SalesItem testSalesItem =
    new SalesItem(1L, 1L, "Test", 10);

  // Create mock implementation of
  // SalesItemRepository interface
  @Mock
  private SalesItemRepository salesItemRepositoryMock;

  // Create mock implementation of
  // SalesItemFactory interface
  @Mock
  private SalesItemFactory salesItemFactoryMock;

  // Injects the above created mocks to salesItemService
  @InjectMocks
  private SalesItemService salesItemService =
    new SalesItemServiceImpl();

  @Test
  final void testCreateSalesItem() {
    // GIVEN
    final var salesItemArg = new SalesItemArg(1L, "Test", 10);

    // Instructs to return 'testSalesItem' when
    // salesItemFactoryMock's createFrom
    // method is called with an argument that reference
    // equals 'salesItemArg'
    Mockito
      .when(salesItemFactoryMock.createFrom(refEq(salesItemArg)))
      .thenReturn(testSalesItem);

    // Instructs to return 'testSalesItem' when
    // salesItemRepositoryMock's 'save' method is called
    // with an argument that reference equals 'testSalesItem'
    Mockito
      .when(salesItemRepositoryMock
        .save(refEq(testSalesItem)))
      .thenReturn(testSalesItem);

    // WHEN
    final var createdSalesItem =
      salesItemService.createSalesItem(salesItemArg);

    // THEN
    assertEquals(createdSalesItem,
                 testSalesItem,
                 SALES_ITEMS_NOT_EQUAL);
  }

  @Test
  final void testGetSalesItems() {
    // GIVEN
```

```
    // Instructs to return a list of containing one sales item
    // 'salesItem' when salesItemRepository's 'findAll'
    // method is called
    Mockito
      .when(salesItemRepositoryMock.findAll())
      .thenReturn(List.of(testSalesItem));

    // WHEN
    final var foundSalesItems = salesItemService.getSalesItems();

    // THEN
     final var iterator = foundSalesItems.iterator();

     assertEquals(iterator.next(),
                  testSalesItem,
                  SALES_ITEMS_NOT_EQUAL);

     assertFalse(iterator.hasNext());
  }
}
```

Java has many testing frameworks and mocking libraries. Below is a small example from a JakartaEE microservice that uses TestNG and JMockit libraries for unit testing and mocking, respectively. In the below example, we are testing a couple of methods from a `ChartStore` class, which is responsible for handling the persistence of chart entities using Java Persistence API (JPA).

ChartStoreTests.java

```
import com.silensoft.conflated...DuplicateEntityError;
import mockit.Expectations;
import mockit.Injectable;
import mockit.Mocked;
import mockit.Tested;
import mockit.Verifications;
import org.testng.annotations.Test;

import javax.persistence.EntityExistsException;
import javax.persistence.EntityManager;

import java.util.Collections;
import java.util.List;

import static org.testng.Assert.assertEquals;
import static org.testng.Assert.fail;

public class ChartStoreTests {
  // chartStore will contain an instance
  // of ChartStoreImpl after @Tested
  // annotation is processed
  @Tested
  private ChartStore chartStore;

  // @Injectable annotation creates a mock instance
  // of EntityManager interface and then injects
  // it where needed
  // In this case, it will be injected to 'chartStore'
  @Injectable
  private EntityManager entityManager;

  // Create a mock instance of Chart
```

```
  // (does not inject anywhere)
  @Mocked
  private Chart chartMock;

  @Test
  void testCreate() {
    // WHEN
    chartStore.create(chartMock);

    // THEN
    // JMockit's verification block checks
    // that below mock functions are called
    new Verifications() {{
      chartStore.getEntityManager().persist(chartMock);
      chartStore.getEntityManager().flush();
    }};
  }

  @Test
  void testCreate_whenChartAlreadyExists() {
    // GIVEN
    // JMockit's expectations block will define what mock methods
    // calls are expected and also can specify
    // the return value or result of the mock method call.
    // Below the 'persist' mock method call will throw
    // EntityExistsException
    new Expectations() {{
      chartStore.getEntityManager().persist(chartMock);
      result = new EntityExistsException();
    }};

    try {
      // WHEN
      chartStore.create(chartMock);
      fail("Expected exception, but it was not thrown");
    } catch (final DuplicateEntityError error) {
      // THEN successfully throws an error
    }
  }

  @Test
  void testGetChartById() {
    // GIVEN
    new Expectations() {{
      chartStore.getEntityManager().find(Chart.class, 1L);
      result = chartMock;
    }};

    // WHEN
    final var chart = chartStore.getById(1L);

    // THEN
    assertEquals(chart, chartMock);
  }
}
```

Let's have a unit testing example with JavaScript/TypeScript. We will write a unit test for the following function using the Jest library:

fetchTodos.ts

```
import store from '../../store/store';
import todoService from '../services/todoService';

export default async function fetchTodos(): Promise<void> {
  const { todosState } = store.getState();
  todosState.isFetching = true;
  try {
    todosState.todos = await todoService.tryFetchTodos();
    todosState.fetchingHasFailed = false;
  } catch(error) {
    todosState.fetchingHasFailed = true;
  }
  todosState.isFetching = false;
}
```

Below is the unit test case for the happy path scenario:

fetchTodos.test.ts

```
import store from '../../store/store';
import todoService from '../services/todoService';
import fetchTodos from 'fetchTodos';
// ...

// Mock both 'store' and 'todoService' objects
jest.mock('../../store/store');
jest.mock('../services/todoService');

describe('fetchTodos', async () => {
  it('should fetch todos from todo service', async () => {
    // GIVEN
    const todosState = { todos: [] } as TodoState;
    store.getState.mockReturnValue({ todosState });

    const todos = [{
      id: 1,
      name: 'todo',
      isDone: false
    }];

    todoService.tryFetchTodos.mockResolvedValue(todos);

    // WHEN
    await fetchTodos();

    // THEN
    expect(todosState.isFetching).toBe(false);
    expect(todosState.fetchingHasFailed).toBe(false);
    expect(todosState.todos).toBe(todos);
  });
});
```

In the above example, we used the `jest.mock` function to create mocked versions of the `store` and `todoService` modules. Another way to handle mocking with Jest is using `jest.fn()`, which creates a mocked function. Let's assume that the `fetchTodos` function is changed so that it takes a `store` and `todoService` as its arguments:

fetchTodos.ts

```
// ...

export default async function fetchTodos(
  store: Store,
  todoService: TodoService
): Promise<void> {
  // Same code here as in earlier example...
}
```

Now the mocking would look like the following:

fetchTodos.test.ts

```
import fetchTodos from 'fetchTodos';
// ...

const store = {
   getState: jest.fn()
};

const todoService = {
   tryFetchTodos: jest.fn();
}

describe('fetchTodos', async () => {
  it('should fetch todos from todo service', async () => {
    // GIVEN
    // Same code as in earlier example...

    // WHEN
    await fetchTodos(store as any, todoService as any);

    // THEN
    // Same code as in earlier example...
  });
});
```

Let's have an example with C++ and Google Test unit testing framework. In C++, you can define a mock class by extending a pure virtual base class ("interface") and using Google Mock macros to define mocked methods. Below is the definition of a `detectedAnomalies` method that we want to unit test:

AnomalyDetectionEngine.h

```
class AnomalyDetectionEngine
{
public:
  virtual ~AnomalyDetectionEngine() = default;

  virtual void detectAnomalies() = 0;
};
```

AnomalyDetectionEngineImpl.h

```
#include <memory>
#include "AnomalyDetectionEngine.h"
#include "Configuration.h"
```

```
class AnomalyDetectionEngineImpl :
  public AnomalyDetectionEngine
{
public:
  explicit AnomalyDetectionEngineImpl(
    std::shared_ptr<Configuration> configuration
  );

  void detectAnomalies() override;

private:
  void detectAnomalies(
    const std::shared_ptr<AnomalyDetectionRule>& anomalyDetectionRule
  );

  std::shared_ptr<Configuration> m_configuration;
};
```

AnomalyDetectionEngineImpl.cpp

```
#include <algorithm>
#include <execution>
#include "AnomalyDetectionEngineImpl.h"

AnomalyDetectionEngineImpl::AnomalyDetectionEngineImpl(
  std::shared_ptr<Configuration> configuration
) : m_configuration(std::move(configuration))
{}

void AnomalyDetectionEngineImpl::detectAnomalies()
{
  const auto anomalyDetectionRules =
    m_configuration->getAnomalyDetectionRules();

   std::for_each(std::execution::par,
                 anomalyDetectionRules->cbegin(),
                 anomalyDetectionRules->cend(),
                 [this](const auto& anomalyDetectionRule)
                 {
                   detectAnomalies(anomalyDetectionRule);
                 });
}

void AnomalyDetectionEngineImpl::detectAnomalies(
  const std::shared_ptr<AnomalyDetectionRule>& anomalyDetectionRule
)
{
  const auto anomalyIndicators = anomalyDetectionRule->detectAnomalies();

  std::ranges::for_each(*anomalyIndicators,
                        [](const auto& anomalyIndicator)
                        {
                          anomalyIndicator->publish();
                        });
}
```

Let's create a `Configuration` class and a `ConfigurationMock` class for mocks:

Configuration.h

```
#include <memory>
#include <vector>
#include "AnomalyDetectionRule.h"

class Configuration
{
public:
  virtual ~Configuration() = default;

  virtual std::shared_ptr<AnomalyDetectionRules>
  getAnomalyDetectionRules() const = 0;
};
```

ConfigurationMock.h

```
#include <gmock/gmock.h>
#include "Configuration.h"

class ConfigurationMock : public Configuration
{
public:
  MOCK_METHOD(std::shared_ptr<AnomalyDetectionRules>,
              getAnomalyDetectionRules, (), (const)
  );
};
```

Let's create an `AnomalyDetectionRule` class and a respective mock class, `AnomalyDetectionRuleMock`:

AnomalyDetectionRule.h

```
#include "AnomalyIndicator.h"

class AnomalyDetectionRule
{
public:
  virtual ~AnomalyDetectionRule() = default;

  virtual std::shared_ptr<AnomalyIndicators>
  detectAnomalies() = 0;
};

using AnomalyDetectionRules =
  std::vector<std::shared_ptr<AnomalyDetectionRule>>;
```

AnomalyDetectionRuleMock.h

```
#include <gmock/gmock.h>
#include "AnomalyDetectionRule.h"

class AnomalyDetectionRuleMock : public AnomalyDetectionRule
{
  public:
    MOCK_METHOD(std::shared_ptr<AnomalyIndicators>,
                detectAnomalies, ());
};
```

Let's create an `AnomalyIndicator` class and a mock class, `AnomalyIndicatorMock`:

AnomalyIndicator.h

```
#include <memory>
#include <vector>

class AnomalyIndicator
{
public:
  virtual ~AnomalyIndicator() = default;

  virtual void publish() = 0;
};

using AnomalyIndicators =
  std::vector<std::shared_ptr<AnomalyIndicator>>;
```

AnomalyIndicatorMock.h

```
#include <gmock/gmock.h>
#include "AnomalyIndicator.h"

class AnomalyIndicatorMock : public AnomalyIndicator
{
public:
  MOCK_METHOD(void, publish, ());
};
```

Let's create a unit test for the `detectAnomalies` method in the `AnomalyDetectionEngineImpl` class:

AnomalyDetectionEngineImplTests.h

```
#include <gtest/gtest.h>
#include "ConfigurationMock.h"
#include "AnomalyDetectionRuleMock.h"
#include "AnomalyIndicatorMock.h"

class AnomalyDetectionEngineImplTests : public testing::Test
{
protected:
  void SetUp() override {
    m_anomalyDetectionRules->push_back(m_anomalyDetectionRuleMock);
    m_anomalyIndicators->push_back(m_anomalyIndicatorMock);
  }

  std::shared_ptr<ConfigurationMock> m_configurationMock{
    std::make_shared<ConfigurationMock>()
  };

  std::shared_ptr<AnomalyDetectionRuleMock> m_anomalyDetectionRuleMock{
    std::make_shared<AnomalyDetectionRuleMock>()
  };

  std::shared_ptr<AnomalyDetectionRules> m_anomalyDetectionRules{
    std::make_shared<AnomalyDetectionRules>()
  };

  std::shared_ptr<AnomalyIndicatorMock> m_anomalyIndicatorMock{
```

```
    std::make_shared<AnomalyIndicatorMock>()
  };

  std::shared_ptr<AnomalyIndicators> m_anomalyIndicators{
    std::make_shared<AnomalyIndicators>()
  }
};
```

AnomalyDetectionEngineImplTests.cpp

```
#include "../src/AnomalyDetectionEngineImpl.h"
#include "AnomalyDetectionEngineImplTests.h"

using testing::Return;

TEST_F(AnomalyDetectionEngineImplTests, testDetectAnomalies)
{
  // GIVEN
  AnomalyDetectionEngineImpl anomalyDetectionEngine{m_configurationMock};

  // EXPECTATIONS
  EXPECT_CALL(*m_configurationMock, getAnomalyDetectionRules)
    .Times(1)
    .WillOnce(Return(m_anomalyDetectionRules));

  EXPECT_CALL(*m_anomalyDetectionRuleMock, detectAnomalies)
    .Times(1)
    .WillOnce(Return(m_anomalyIndicators));

  EXPECT_CALL(*m_anomalyIndicatorMock, publish).Times(1);

  // WHEN
  anomalyDetectionEngine.detectAnomalies();
}
```

The above example did not contain dependency injection, so let's have another example in C++ where dependency injection is used. First, we define a generic base class for singletons:

Singleton.h

```
#include <memory>

template<typename T>
class Singleton
{
public:
  Singleton() = default;

  virtual ~Singleton()
  {
    m_instance.reset();
  };

  static inline std::shared_ptr<T>& getInstance()
  {
    return m_instance;
  }

  static void setInstance(const std::shared_ptr<T>& instance)
  {
    m_instance = instance;
```

```
  }

private:
  static inline std::shared_ptr<T> m_instance;
};
```

Next, we implement a configuration parser that we will later unit test:

ConfigParserImpl.h

```
#include <memory>
#include "Configuration.h"

class ConfigParserImpl {
public:
  std::shared_ptr<Configuration> parse();
};
```

ConfigParserImpl.cpp

```
#include "AnomalyDetectionRulesParser.h"
#include "Configuration.h"
#include "ConfigFactory.h"
#include "ConfigParserImpl.h"
#include "MeasurementDataSourcesParser.h"

std::shared_ptr<Configuration>
ConfigParserImpl::parse(...)
{
  const auto measurementDataSources =
    MeasurementDataSourcesParser::getInstance()->parse(...);

  const auto anomalyDetectionRules =
    AnomalyDetectionRulesParser::getInstance()->parse(...);

  return ConfigFactory::getInstance()
    ->createConfig(anomalyDetectionRules);
}
```

Next, we define `MeasurementDataSource`, `MeasurementDataSourcesParser`, and `MeasurementDataSourcesParserImpl` classes:

MeasurementDataSource.h

```
#include <memory>
#include <vector>

class MeasurementDataSource {
  // ...
};

using MeasurementDataSources =
  std::vector<std::shared_ptr<MeasurementDataSource>>;
```

MeasurementDataSourcesParser.h

```
#include "Singleton.h"
#include "MeasurementDataSource.h"
```

```
class MeasurementDataSourcesParser :
  public Singleton<MeasurementDataSourcesParser>
{
public:
  virtual std::shared_ptr<MeasurementDataSources> parse(...) = 0;
};
```

MeasurementDataSourcesParserImpl.h

```
#include "MeasurementDataSourcesParser.h"

class MeasurementDataSourcesParserImpl :
  public MeasurementDataSourcesParser
{
public:
  std::shared_ptr<MeasurementDataSources> parse(...) override {
    // ...
  }
};
```

Next, we define `AnomalyDetectionRulesParser` and `AnomalyDetectionRulesParserImpl` classes:

AnomalyDetectionRulesParser.h

```
#include "Singleton.h"
#include "AnomalyDetectionRule.h"

class AnomalyDetectionRulesParser :
  public Singleton<AnomalyDetectionRulesParser>
{
public:
  virtual std::shared_ptr<AnomalyDetectionRules> parse(...) = 0;
};
```

AnomalyDetectionRulesParserImpl.h

```
#include "AnomalyDetectionRulesParser.h"

class AnomalyDetectionRulesParserImpl :
  public AnomalyDetectionRulesParser
{
public:
  std::shared_ptr<AnomalyDetectionRules> parse(...) override {
   // ...
  }
};
```

Next, we define `ConfigFactory` and `ConfigFactoryImpl` classes:

ConfigFactory.h

```
#include "Singleton.h"
#include "Configuration.h"

class ConfigFactory :
  public Singleton<ConfigFactory>
{
public:
```

```
  virtual std::shared_ptr<Configuration>
  createConfig(
    const std::shared_ptr<AnomalyDetectionRules>& rules
  ) = 0;
};
```

ConfigFactoryImpl.h

```
#include "ConfigFactory.h"

class ConfigFactoryImpl : public ConfigFactory
{
public:
  std::shared_ptr<Configuration>
  createConfig(
    const std::shared_ptr<AnomalyDetectionRules>& rules
  ) override {
    // ...
  }
};
```

Then we define a dependency injector class:

DependencyInjector.h

```
#include "AnomalyDetectionRulesParserImpl.h"
#include "ConfigFactoryImpl.h"
#include "MeasurementDataSourcesParserImpl.h"

class DependencyInjector final
{
public:
  static void injectDependencies()
  {
    AnomalyDetectionRulesParser::setInstance(
      std::make_shared<AnomalyDetectionRulesParserImpl>()
    );

    ConfigFactory::setInstance(
      std::make_shared<ConfigFactoryImpl>()
    );

    MeasurementDataSourcesParser::setInstance(
      std::make_shared<MeasurementDataSourcesParserImpl>()
    );
  }

private:
  DependencyInjector() = default;
};
```

We inject dependencies upon application startup using the dependency injector:

main.cpp

```
#include "DependencyInjector.h"

int main()
{
  DependencyInjector::injectDependencies();
```

```
  // Initialize and start application...
}
```

Let's define a unit test class for the `ConfigParserImpl` class:

ConfigParserImplTests.h

```
#include "MockDependenciesInjectedTest.h"

class ConfigParserImplTests :
  public MockDependenciesInjectedTest
{};
```

All unit test classes should inherit from a base class that injects mock dependencies. When tests are completed, the mock dependencies will be removed. The Google Test framework requires this removal because it only validates expectations on a mock upon the mock object destruction.

MockDependenciesInjectedTest.h

```
#include <gtest/gtest.h>
#include "MockDependencyInjector.h"

class MockDependenciesInjectedTest :
  public testing::Test
{
protected:
  void SetUp() override
  {
    m_mockDependencyInjector.injectMockDependencies();
  }

  void TearDown() override
  {
    m_mockDependencyInjector.removeMockDependencies();
  }

  MockDependencyInjector m_mockDependencyInjector;
};
```

Below are all the mock classes defined:

AnomalyDetectionRulesParserMock.h

```
#include <gmock/gmock.h>
#include "AnomalyDetectionRulesParser.h"

class AnomalyDetectionRulesParserMock :
  public AnomalyDetectionRulesParser
{
public:
  MOCK_METHOD(std::shared_ptr<AnomalyDetectionRules>, parse, (...));
};
```

ConfigFactoryMock.h

```
#include <gmock/gmock.h>
#include "ConfigFactory.h"
```

```
class ConfigFactoryMock : public ConfigFactory
{
public:
  MOCK_METHOD(
    std::shared_ptr<Configuration>,
    createConfig,
    (const std::shared_ptr<AnomalyDetectionRules>& rules)
  );
};
```

MeasurementDataSourcesParserMock.h

```
#include <gmock/gmock.h>
#include "MeasurementDataSourcesParser.h"

class MeasurementDataSourcesParserMock :
  public MeasurementDataSourcesParser
{
public:
  MOCK_METHOD(std::shared_ptr<MeasurementDataSources>, parse, (...));
};
```

Below is the `MockDependencyInjector` class defined:

MockDependencyInjector.h

```
#include "AnomalyDetectionRulesParserMock.h"
#include "ConfigFactoryMock.h"
#include "MeasurementDataSourcesParserMock.h"

class MockDependencyInjector final
{
public:
  std::shared_ptr<AnomalyDetectionRulesParserMock>
  m_anomalyDetectionRulesParserMock{
    std::make_shared<AnomalyDetectionRulesParserMock>()
  };

  std::shared_ptr<ConfigFactoryMock> m_configFactoryMock{
    std::make_shared<ConfigFactoryMock>()
  };

  std::shared_ptr<MeasurementDataSourcesParserMock>
  m_measurementDataSourcesParserMock{
    std::make_shared<MeasurementDataSourcesParserMock>()
  };

  void injectMockDependencies() const
  {
    AnomalyDetectionRulesParser::setInstance(
      m_anomalyDetectionRulesParserMock
    );

    ConfigFactory::setInstance(
      m_configFactoryMock
    );

    MeasurementDataSourcesParser::setInstance(
      m_measurementDataSourcesParserMock
    );
```

```
  }

  void removeMockDependencies() const {
    AnomalyDetectionRulesParser::setInstance({nullptr});
    ConfigFactory::setInstance({nullptr});
    MeasurementDataSourcesParser::setInstance({nullptr});
  }
};
```

Below is the unit test implementation that uses the mocks:

ConfigParserImplTests.cpp

```
#include "ConfigParserImplTests.h"
#include "ConfigParserImpl.h"

using testing::Eq;
using testing::Return;

TEST_F(ConfigParserImplTests, testParseConfig)
{
  // GIVEN
  ConfigParserImpl configParser;

  // EXPECTATIONS
  EXPECT_CALL(
    *m_mockDependencyInjector.m_anomalyDetectionRulesParserMock,
    parse
  ).Times(1)
   .WillOnce(Return(m_anomalyDetectionRules));

  EXPECT_CALL(
    *m_mockDependencyInjector.m_measurementDataSourcesParserMock,
    parse
  ).Times(1)
   .WillOnce(Return(m_measurementDataSources));

  EXPECT_CALL(
    *m_mockDependencyInjector.m_configFactoryMock,
    createConfig(Eq(m_anomalyDetectionRules))
  ).Times(1)
   .WillOnce(Return(m_configMock));

  // WHEN
  const auto configuration = configParser.parse();

  // THEN
  ASSERT_EQ(configuration, m_configMock);
}
```

You can also make sure that implementation class instances can be created only in the `DependencyInjector` class by declaring implementation class constructors private and making the `DependencyInjector` class a friend of the implementation classes. In this way, no one can accidentally create an instance of an implementation class. Instances of implementation classes should be created by the dependency injector only. Below is a implementation class where the constructor is made private, and the dependency injector is made a friend of the class:

AnomalyDetectionRulesParserImpl.h

```
#include "AnomalyDetectionRulesParser.h"

class AnomalyDetectionRulesParserImpl :
  public AnomalyDetectionRulesParser
{
  friend class DependencyInjector;

public:
  std::shared_ptr<AnomalyDetectionRules> parse() override;

private:
  AnomalyDetectionRulesParserImpl() = default;
};
```

UI Component Unit Testing

UI component unit testing differs from regular unit testing because you cannot necessarily test the functions of a UI component in isolation if you have, for example, a React functional component. You must conduct UI component unit testing by mounting the component to DOM and then perform tests by triggering events, for example. This way, you can test the event handler functions of a UI component. The rendering part should also be tested. It can be tested by producing a snapshot of the rendered component and storing that in version control. Further rendering tests should compare the rendered result to the snapshot stored in the version control.

Below is an example of testing the rendering of a React component, `NumberInput`:

NumberInput.test.js

```
import renderer from 'react-test-renderer';
// ...

describe('NumberInput') () => {
  // ...

  describe('render', () => {
    it('renders with buttons on left and right"', () => {
      const numberInputAsJson =
        renderer
          .create(<NumberInput buttonPlacement="leftAndRight"/>)
          .toJSON();

      expect(numberInputAsJson).toMatchSnapshot();
    });

    it('renders with buttons on right', () => {
      const numberInputAsJson =
        renderer
          .create(<NumberInput buttonPlacement="right"/>)
          .toJSON();

      expect(numberInputAsJson).toMatchSnapshot();
    });
  });
});
```

Below is an example unit test for the number input's decrement button's click event handler

function, `decrementValue`:

NumberInput.test.js

```
import { render, fireEvent, screen } from '@testing-library/react'
// ...

describe('NumberInput') () => {
  // ...

  describe('decrementValue', () => {
    it('should decrement value by given step amount', () => {
      render(<NumberInput value="3" stepAmount={2} />);
      fireEvent.click(screen.getByText('-'));
      const numberInputElement = screen.getByDisplayValue('1');
      expect(numberInputElement).toBeTruthy();
    });
  });
});
```

In the above example, we used the *testing-library*, which has implementations for all the common UI frameworks: React, Vue and Angular. It means you can use mostly the same testing API regardless of your UI framework. There are tiny differences, basically only in the syntax of the `render` method. If you had implemented some UI components and unit tests for them with React, and you would like to reimplement them with Vue, you don't need to reimplement all the unit tests. You only need to modify them slightly (e.g., make changes to the `render` function calls). Otherwise, the existing unit tests should work because the behavior of the UI component did not change, only its internal implementation from React to Vue.

Software Component Integration Testing Principle

> *Integration testing aims to test that a software component works against actual dependencies and that its public methods correctly understand the purpose and signature of other public methods they are using.*

In the software component integration testing, all public functions of a software component should be tested. Not all functionality of the public functions should be tested because that has already been done in the unit testing phase. This is why there are fewer integration tests than unit tests. The term *integration testing* sometimes refers to the integration of a complete software system or a product. However, it should be used to describe software component integration only. When testing a product or a software system, the term *E2E testing* should be used to avoid confusion and misunderstandings.

The best way to define integration tests is by using *behavior-driven development* (BDD). BDD encourages teams to use domain-driven design and concrete examples to formalize a shared understanding of how a software component should behave. In BDD, behavioral specifications are the root of the integration tests. A team can create behavioral specifications during the initial domain-driven design phase. This practice will shift the integration testing to the left, meaning that

writing the integration tests starts early and can proceed in parallel with the actual implementation. One widely used and recommended way to write behavioral specifications is the *Gherkin* language.

When using the Gherkin language, the behavior of a software component is described as features. There should be a separate file for each feature. These files have the *.feature* extension. Each feature file describes one feature and one or more scenarios for that feature. The first scenario should be the so-called "happy path" scenario, and other possible scenarios should handle additional happy paths, failures, and edge cases that need to be tested. Remember that you don't have to test every failure and edge case because those were already tested in the unit testing phase.

Below is a simplified example of one feature in a *data-visualization-configuration-service*. The service is a REST API. The feature is for creating a new chart. (In a real-life scenario, a chart contains more properties like the chart's data source and what measure(s) and dimension(s) are shown in the chart, for example)

createChart.feature

```
Feature: Create chart
  Creates a new chart

  Scenario: Creates a new chart successfully
    Given chart layout id is "1"
    And chart type is "line"
    And X-axis categories shown count is 10
    And fetchedRowCount is 1000

    When I create a new chart

    Then I should get the chart given above
         with response code 201 "Created"
```

The above example shows how the feature's name is given after the `Feature` keyword. You can add free-form text below the feature's name to describe the feature in more detail. Next, a scenario is defined after the `Scenario` keyword. First, the name of the scenario is given. Then comes the steps of the scenario. Each step is defined using one of the following keywords: `Given`, `When`, `Then`, `And`, and `But`. A scenario should follow this pattern:

- Steps to describe initial context/setup (`Given/And` steps)
- Steps to describe an event (`When` step)
- Steps to describe the expected outcome for the event (`Then/And` steps)

We can add another scenario to the above example:

createChart.feature

```
Feature: Create chart
  Creates a new chart

  Scenario: Creates a new chart successfully
    Given chart layout id is "1"
```

```
    And chart type is "line"
    And X-axis categories shown count is 10
    And fetchedRowCount is 1000

    When I create a new chart

    Then I should get the chart given above
         with status code 201 "Created"

  Scenario: Chart creation fails due to missing mandatory parameter
    When I create a new chart

    Then I should get a response with status code 400 "Bad Request"
    And response body should contain "is mandatory field" entry
        for following fields
      | layoutId                  |
      | fetchedRowCount           |
      | xAxisCategoriesShownCount |
      | type                      |
```

Now we have one feature with two scenarios specified. Next, we shall implement the scenarios. Our *data-visualization-configuration-service* is implemented in Java, and we want to implement also the integration tests in Java. *Cucumber* (https://cucumber.io/docs/installation/) has BDD tools for various programming languages. We will be using the *Cucumber-JVM* (https://cucumber.io/docs/installation/java/) library.

We place integration test code into the source code repository's *src/test* directory. The feature files are in the *src/test/resources/features* directory. Feature directories should be organized into subdirectories in the same way source code is organized into subdirectories: using domain-driven design and creating subdirectories for subdomains. We can put the above *createChart.feature* file to the *src/test/resources/features/chart* directory.

Next, we need to provide an implementation for each step in the scenarios. Let's start with the first scenario. We shall create a file *TestContext.java* for the test context and a *CreateChartStepDefs.java* file for the step definitions:

TestContext.java

```
public class TestContext {
  public io.restassured.response.Response response;
}
```

CreateChartStepDefs.java

```
import integrationtests.TestContext;
import com.silensoft.dataviz.configuration.service.chart.Chart;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import io.restassured.http.ContentType;
import io.restassured.mapper.ObjectMapperType;

import static io.restassured.RestAssured.given;
import static org.hamcrest.Matchers.equalTo;
import static org.hamcrest.Matchers.greaterThan;
```

```
public class CreateChartStepDefs {
  private static final String BASE_URL =
    "http://localhost:8080/data-visualization-configuration-service/";

  private final TestContext testContext;
  private final Chart chart = new Chart();

  public CreateChartStepDefs(final TestContext testContext) {
    this.testContext = testContext;
  }

  @Given("chart layout id is {string}")
  public void setChartLayoutId(final String layoutId) {
    chart.setLayoutId(layoutId);
  }

  @Given("chart type is {string}")
  public void setChartType(final String chartType) {
    chart.setType(chartType);
  }

  @Given("X-axis categories shown count is {int}")
  public void setXAxisCategoriesShownCount(
    final Integer xAxisCategoriesShownCount
  ) {
    chart
      .setxAxisCategoriesShownCount(xAxisCategoriesShownCount);
  }

  @Given("fetchedRowCount is {int}")
  public void setFetchedRowCount(final Integer fetchedRowCount) {
    chart.setFetchedRowCount(fetchedRowCount);
  }

  @When("I create a new chart")
  public void createNewChart() {
    testContext.response = given()
      .contentType("application/json")
      .body(chart, ObjectMapperType.GSON)
      .when()
      .post(Constants.BASE_URL + "charts");
  }

   @Then("I should get the chart given above with status code {int} {string}")
   public void iShouldGetTheChartGivenAbove(
     final int statusCode,
     final String statusCodeName
   ) {
     testContext.response.then()
       .assertThat()
       .statusCode(statusCode)
       .body("id", greaterThan(0))
       .body("layoutId", equalTo(chart.getLayoutId()))
       .body("type", equalTo(chart.getType()))
       .body("xAxisCategoriesShownCount",
             equalTo(chart.getXAxisCategoriesShownCount()))
       .body("fetchedRowCount",
              equalTo(chart.getFetchedRowCount()));
 }
}
```

The above implementation contains a function for each step. Each function is annotated with an annotation for a specific Gherkin keyword: `@Given`, `@When`, and `@Then`. Note that a step in a scenario can be templated. For example, the step `Given chart layout id is "1"` is templated and defined in the function `@Given("chart layout id is {string}") public void setChartLayoutId(final String layoutId)` where the actual layout id is given as a parameter to the function. You can use this templated step in different scenarios that can give a different value for the layout id, for example: `Given chart layout id is "8"`.

The `createNewChart` method uses *REST-assured* (https://rest-assured.io/) for submitting an HTTP POST request to the *data-visualization-configuration-service*. And the `iShouldGetTheChartGivenAbove` function takes the HTTP POST response and validates the status code and the properties in the response body.

The second scenario is a common failure scenario where you create something with missing parameters. Because this scenario is common (i.e., we can use the same steps in other features), we put the step definitions in a file named *CommonStepDefs.java* in the *common* subdirectory of the *src/test/java/integrationtests* directory.

Here are the step definitions:

CommonStepDefs.java

```
import integrationtests.TestContext;
import io.cucumber.java.en.And;
import io.cucumber.java.en.Then;

import java.util.List;

import static org.hamcrest.Matchers.hasItems;

public class CommonStepDefs {
  private final TestContext testContext;

  public CommonStepDefs(final TestContext testContext) {
    this.testContext = testContext;
  }

  @Then("I should get a response with status code {int} {string}")
  public void iShouldGetAResponseWithResponseCode(
    final int statusCode,
    final String statusCodeName
  ) {
      testContext.response.then()
        .assertThat()
        .statusCode(statusCode);
  }

  @And("response body should contain {string} entry for following fields")
  public void responseBodyShouldContainEntryForFollowingFields(
    final String entry,
    final List<String> fields
  ) {
    testContext.response.then()
      .assertThat()
```

```
      .body("", hasItems(fields
                          .stream()
                          .map(field -> field + ' ' + entry)
                          .toArray()));
  }
}
```

Cucumber is available in many other languages in addition to Java. It is available, for example, for JavaScript (Cucumber.js) and Python (Behave, https://behave.readthedocs.io/en/stable/). Integration tests can be written in a different language than the language used for implementation and unit test code. For example, I am currently developing a microservice in C++. Our team has a test automation developer working with integration tests using the Gherkin language for feature definitions and Python and Behave for implementing the steps.

Some frameworks offer their way of creating integration tests. The Spring Boot framework offers the `MockMvc` to test the web layer, like a REST API. Below is an example that uses the `MockMvc` for testing a *sales-item-service* REST API:

SalesItemControllerTests.java

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.MethodOrderer;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.delete;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.put;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
@ExtendWith(SpringExtension.class)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class SalesItemControllerTests {
  private static final long SALES_ITEM_USER_ACCOUNT_ID = 1L;
  private static final String SALES_ITEM_NAME = "Test sales item";
  private static final int SALES_ITEM_PRICE = 10;
  private static final int UPDATED_SALES_ITEM_PRICE = 20;

  private static final String UPDATED_SALES_ITEM_NAME =
    "Updated test sales item";

  @Autowired
  private MockMvc mockMvc;

  @Test
```

```
@Order(1)
final void testCreateSalesItem() throws Exception {
  // GIVEN
  final var salesItemArg =
    new SalesItemArg(SALES_ITEM_USER_ACCOUNT_ID,
                     SALES_ITEM_NAME,
                     SALES_ITEM_PRICE);

  final var salesItemArgJson =
    new ObjectMapper().writeValueAsString(salesItemArg);

  // WHEN
  mockMvc
    .perform(post(SalesItemController.API_ENDPOINT)
               .contentType(MediaType.APPLICATION_JSON)
               .content(salesItemArgJson))
    .andDo(print())
    // THEN
    .andExpect(jsonPath("$.id").value(1))
    .andExpect(jsonPath("$.name").value(SALES_ITEM_NAME))
    .andExpect(jsonPath("$.price").value(SALES_ITEM_PRICE))
    .andExpect(status().isCreated());
}

@Test
@Order(2)
final void testGetSalesItems() throws Exception {
  // WHEN
  mockMvc
    .perform(get(SalesItemController.API_ENDPOINT))
    .andDo(print())
    // THEN
    .andExpect(jsonPath("$[0].id").value(1))
    .andExpect(jsonPath("$[0].name").value(SALES_ITEM_NAME))
    .andExpect(status().isOk());
}

@Test
@Order(3)
final void testGetSalesItemById() throws Exception {
  // WHEN
  mockMvc
    .perform(get(SalesItemController.API_ENDPOINT + "/1"))
    .andDo(print())
    // THEN
    .andExpect(jsonPath("$.name").value(SALES_ITEM_NAME))
    .andExpect(status().isOk());
}

@Test
@Order(4)
final void testGetSalesItemsByUserAccountId() throws Exception {
  // GIVEN
  final var url = SalesItemController.API_ENDPOINT +
                  "?userAccountId=" + SALES_ITEM_USER_ACCOUNT_ID;

  // WHEN
  mockMvc
    .perform(get(url))
    .andDo(print())
    // THEN
    .andExpect(jsonPath("$[0].name").value(SALES_ITEM_NAME))
    .andExpect(status().isOk());
```

```
}

@Test
@Order(5)
final void testUpdateSalesItem() throws Exception {
  // GIVEN
  final var salesItemArg =
    new SalesItemArg(SALES_ITEM_USER_ACCOUNT_ID,
                     UPDATED_SALES_ITEM_NAME,
                     UPDATED_SALES_ITEM_PRICE);

  final var salesItemArgJson =
    new ObjectMapper().writeValueAsString(salesItemArg);

  // WHEN
  mockMvc
    .perform(put(SalesItemController.API_ENDPOINT + "/1")
               .contentType(MediaType.APPLICATION_JSON)
               .content(salesItemArgJson))
    .andDo(print());

  // THEN
  mockMvc
    .perform(get(SalesItemController.API_ENDPOINT + "/1"))
    .andDo(print())
    .andExpect(jsonPath("$.name").value(UPDATED_SALES_ITEM_NAME))
    .andExpect(jsonPath("$.price").value(UPDATED_SALES_ITEM_PRICE))
    .andExpect(status().isOk());
}

@Test
@Order(6)
final void testDeleteSalesItemById() throws Exception {
  // WHEN
  mockMvc
    .perform(delete(SalesItemController.API_ENDPOINT + "/1"))
    .andDo(print());

  // THEN
  mockMvc
    .perform(get(SalesItemController.API_ENDPOINT + "/1"))
    .andDo(print())
    .andExpect(status().isNotFound());
}

@Test
@Order(7)
final void testDeleteSalesItems() throws Exception {
  // GIVEN
  final var salesItemArg =
    new SalesItemArg(SALES_ITEM_USER_ACCOUNT_ID,
                     SALES_ITEM_NAME,
                     SALES_ITEM_PRICE);

  final var salesItemArgJson =
    new ObjectMapper().writeValueAsString(salesItemArg);

  mockMvc
    .perform(post(SalesItemController.API_ENDPOINT)
               .contentType(MediaType.APPLICATION_JSON)
               .content(salesItemArgJson))
    .andDo(print());
```

```
    // WHEN
    mockMvc
      .perform(delete(SalesItemController.API_ENDPOINT))
      .andDo(print());

    // THEN
    mockMvc
      .perform(get(SalesItemController.API_ENDPOINT))
      .andDo(print())
      .andExpect(jsonPath("$").isEmpty())
      .andExpect(status().isOk());
  }
}
```

If you have an API microservice, one more alternative to implement integration tests is an API development platform like *Postman* (https://www.postman.com/). Postman can be used to write integration tests using JavaScript.

Below is an example API request for creating a new sales item. You can define this in Postman as a new request:

```
POST http://localhost:3000/sales-item-service/sales-items
{
  "name": "Test sales item",
  "price": 10,
}
```

Here is a Postman test case to validate the response to the above request:

```
pm.test("Status code is 201 Created", function () {
  pm.response.to.have.status(201);
});

const salesItem = pm.response.json();
pm.collectionVariables.set("salesItemId", salesItem.id)

pm.test("Sales item name", function () {
  return pm.expect(salesItem.name).to.eql("Test sales item");
})

pm.test("Sales item price", function () {
  return pm.expect(salesItem.price).to.eql(10);
})
```

In the above test case, the response status code is verified first, and then the `salesItem` object is parsed from the response body. Value for the variable `salesItemId` is set. This variable will be used in subsequent test cases. Finally, the values of the `name` and `price` properties are checked.

Next, a new API request could be created in Postman to retrieve the just created sales item:

```
GET http://localhost:3000/sales-item-service/sales-items/{{salesItemId}}
```

We used the value stored in the `salesItemId` variable in the request URL. Variables can be used

in the URL and request body using the following notation: `{{<variable-name>}}`. Let's create a test case for the above request:

```
pm.test("Status code is 200 OK", function () {
  pm.response.to.have.status(200);
});

const salesItem = pm.response.json();

pm.test("Sales item name", function () {
  return pm.expect(salesItem.name).to.eql("Test sales item");
})

pm.test("Sales item price", function () {
  return pm.expect(salesItem.price).to.eql(10);
})
```

API integration tests written in Postman can be utilized in a CI/CD pipeline. An easy way to do that is to export a Postman collection to a file that contains all the API requests and related tests. A Postman collection file is a JSON file. Postman offers a Node.js command-line utility called *Newman* (https://learning.postman.com/docs/running-collections/using-newman-cli/installing-running-newman/). It can be used to run API requests and related tests in an exported Postman collection file.

You can run integration tests in an exported Postman collection file with the below command in a CI/CD pipeline:

```
newman run integration-tests/integrationTestsPostmanCollection.json
```

In the above example, we assume that a file named *integrationTestsPostmanCollection.json* has been exported to the *integration-tests* directory in the source code repository.

UI Integration Testing

You can also use the Gherkin language when specifying UI features. For example, the *TestCafe* UI testing tool can be used with the *gherkin-testcafe* tool to make TestCafe support the Gherkin syntax. Let's create a simple UI feature:

greetUser.feature

```
Feature: Greet user
  Entering user name and clicking submit button
  displays a greeting for the user

  Scenario: Greet user successfully
    Given there is "John Doe" entered in the input field
    When I press the submit button
    Then I am greeted with text "Hello, John Doe"
```

Next, we can implement the above steps in JavaScript using the TestCafe testing API:

GreetUserSteps.js

```
// Imports...

// 'Before' hook runs before the first step of each scenario.
// 't' is the TestCafe test controller object
Before('Navigate to application URL', async (t) => {
  // Navigate browser to application URL
  await t.navigateTo('...');
});

Given('there is {string} entered in the input field',
      async (t, [userName]) => {
  // Finds an HTML element with CSS id selector and
  // enters text to it
  await t.typeText('#user-name', userName);
});

When('I press the submit button', async (t) => {
  // Finds an HTML element with CSS id selector and clicks it
  await t.click('#submit-button');
});

When('I am greeted with text {string}', async (t, [greeting]) => {
  // Finds an HTML element with CSS id selector
  // and compares its inner text
  await t.expect(Selector('#greeting').innerText).eql(greeting);
});
```

There is another similar tool to TestCafe, namely *Cypress*. You can also use Gherkin with Cypress with the *cypress-cucumber-preprocessor* package. Then you can write your UI integration tests like this:

visitDuckDuckGoWebSite.feature

```
Feature: Visit duckduckgo.com website

  Scenario: Visit duckduckgo.com website successfully
    When I visit duckduckgo.com
    Then I should see the search bar
```

VisitDuckDuckGoWebSiteSteps.js

```
import { When, Then } from
  '@badeball/cypress-cucumber-preprocessor';

When("I visit duckduckgo.com", () => {
  cy.visit("https://www.duckduckgo.com");
});

Then("I should see the search bar", () => {
  cy.get("input").should(
    "have.attr",
    "placeholder",
    "Search the web without being tracked"
  );
});
```

Setting Up Integration Testing Environment

Before integration tests can be run, an integration testing environment must be set up. An integration testing environment is where the tested microservice and all its dependencies are running. The easiest way to set up an integration testing environment for a containerized microservice is to use *Docker Compose*, a simple container orchestration tool for a single host.

Let's create a *docker-compose.yml* file for the *sales-item-service*, which has a MySQL database as a dependency.

docker-compose.yml

```
version: "3.8"

services:
  wait-for-services-ready:
    image: dokku/wait
  sales-item-service:
    restart: always
    build:
      context: .
    env_file: .env.ci
    ports:
      - "3000:3000"
    depends_on:
      - mysql
  mysql:
    image: mysql:8.0.22
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    cap_add:
      - SYS_NICE
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
    ports:
      - "3306:3306"
```

In the above example, we first define a service *wait-for-services-ready* which we will use later. Next, we define our microservice, *sales-item-service*. We ask Docker Compose to build a container image for the *sales-item-service* using the *Dockerfile* in the current directory. Then we define the environment for the microservice to be read from an *.env.ci* file. We expose port 3000 and tell that our microservice depends on the *mysql* service.

Next, we define the *mysql* service. We tell what image to use, give a command-line parameter and define the environment and expose a port.

Before we can run the integration tests, we must spin the integration testing environment up using the `docker-compose up` command:

```
docker-compose up --env-file .env.ci --build -d
```

We tell the `docker-compose` command to read environment variables from an *.env.ci* file, which should contain an environment variable named `MYSQL_PASSWORD`. We ask `docker-compose` to always build the *sales-item-service* by specifying the `--build` flag. The `-d` flag tells `docker-`

`compose` to run in the background.

Before we can run the integration tests, we must wait until all services defined in the *docker-compose.yml* are up and running. We use the *wait-for-services-ready* service provided by the *dokku/wait* (https://hub.docker.com/r/dokku/wait) image. We can wait for the services to be ready by issuing the following command:

```
docker-compose
  --env-file .env.ci
  run wait-for-services-ready
  -c mysql:3306,sales-item-service:3000
  -t 600
```

The above command will finish after *mysql* service's port 3306 and *sales-item-service's* port 3000 can be connected. After the above command is finished, you can run the integration tests. In the below example, we run the integration tests using the *newman* CLI tool:

```
newman run integration-tests/integrationTestsPostmanCollection.json
```

After integration tests are completed, you can shut down the integration testing environment:

```
docker-compose down
```

If you need other dependencies in your integration testing environment, you can add them to the *docker-compose.yml* file. If you need to add other microservices with dependencies, you must also add transitive dependencies. For example, if you needed to add another microservice that uses a PostgreSQL database, you would need to add both the other microservice and PostgreSQL database to the *docker-compose.yml* file.

Let's say the *sales-item-service* depends on Apache Kafka 2.x that depends on a Zookeeper service. The *sales-item-service's docker-compose.yml* looks like the below after adding Kafka and Zookeeper:

docker-compose.yml

```
version: "3.8"

services:
  wait-for-services-ready:
    image: dokku/wait
  sales-item-service:
    restart: always
    build:
      context: .
    env_file: .env.ci
    ports:
      - 3000:3000
    depends_on:
      - mysql
      - kafka
  mysql:
```

```
    image: mysql:8.0.22
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    cap_add:
      - SYS_NICE
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
    ports:
      - "3306:3306"
  zookeeper:
    image: bitnami/zookeeper:3.7
    volumes:
      - "zookeeper_data:/bitnami"
    ports:
      - 2181:2181"
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
  kafka:
    image: bitnami/kafka:2.8.1
    volumes:
      - "kafka_data:/bitnami"
    ports:
      - "9092:9092"
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper

volumes:
  zookeeper_data:
    driver: local
  kafka_data:
    driver: local
```

End-to-End (E2E) Testing Principle

> *End-to-end (E2E) testing should test a complete software system (i.e., the integration of microservices) so that each test case is end-to-end (from the software system's south-bound interface to the software system's north-bound interface).*

As the name says, in E2E testing, test cases should be end-to-end. They should test that each microservice is deployed correctly to the test environment and connected to its dependent services. The idea of E2E test cases is not to test details of microservices' functionality because that has already been tested as part of unit and software component integration testing.

Let's consider a telecom network analytics software system that consists of the following applications:

- Data ingestion
- Data correlation
- Data aggregation
- Data exporter

- Data visualization

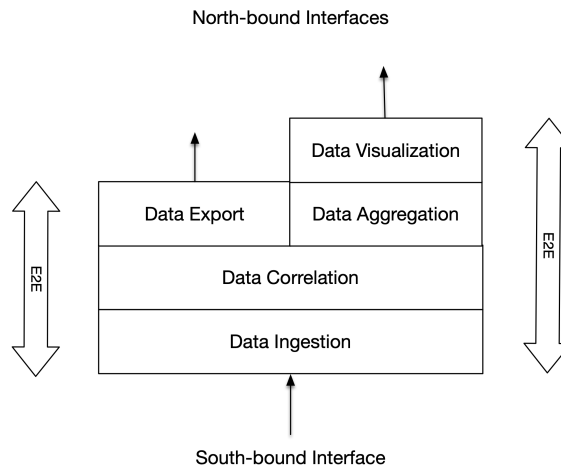

Figure 5.2 Telecom Network Analytics Software System

The southbound interface of the software system is the data ingestion application. The data visualization application provides a web client as a northbound interface. Additionally, the data exporter application provides another northbound interface for the software system.

E2E tests are designed and implemented similarly to software component integration tests. We are just integrating different things (microservices instead of functions). E2E testing should start with the specification of E2E features. These features can be specified using the Gherkin language and put in *.feature* files.

You can start specifying and implementing E2E tests right after the architectural design for the software system is completed. This way, you can shift the implementation of the E2E test to the left and speed up the development phase. You should not start specifying and implementing E2E only when the whole software system is implemented.

Our example software system should have at least two happy-path E2E features. One for testing the data flow from data ingestion to data visualization and another feature to test the data flow from data ingestion to data export. Below is the specification of the first E2E feature:

dataVisualization.feature

```
Feature: Visualize ingested, correlated and
         aggregated data in web UI's dashboard's charts

  Scenario: Data ingested, correlated and aggregated is visualized
            successfully in web UI's dashboard's charts
```

```
   Given southbound interface simulator is configured
         to send input messages that contain data...
   And data ingester is configured to read the input messages
       from the southbound interface
   And data correlator is configured to correlate
       the input messages
   And data aggregator is configured to calculate
       the following counters...
   And data visualization is configured with a dashboard containing
       the following charts viewing the following counters/KPIs...

   When southbound interface simulator sends the input messages
   And data aggregation period is waited
   And data content of each data visualization web UI's dasboard's
       chart is exported to a CSV file

   Then the CSV export file of the first chart should
        contain following values...
   And the CSV export file of the second chart should
        contain following values...
   .
   .
   .
   And the CSV export file of the last chart should
        contain following values...
```

And then, we can create the other feature that tests the E2E path from data ingestion to data export:

dataExport.feature

```
Feature: Export ingested, correlated and transformed data
         to Apache Pulsar

  Scenario: Data ingested, correlated and transformed is
            successfully exported to Apache Pulsar
    Given southbound interface simulator is configured to send
          input messages that contain data...
    And data ingester is configured to read the input messages
        from the southbound interface
    And data correlator is configured to correlate
       the input messages
    And data exporter is configured to export messages with
        the following transformations to Apache Pulsar...

    When southbound interface simulator sends the input messages
    And messages from Apache Pulsar are consumed

    Then first message from Apache Pulsar should have
         the following fields with following values...
    And second message from Apache Pulsar should have
        the following fields with following values...
    .
    .
    .
    And last message from Apache Pulsar should have
        the following fields with following values...
```

Next, E2E tests can be implemented. Any programming language and tool compatible with the Gherkin syntax, like Behave with Python, can be used.

The software system we want to E2E test must reside in a production-like test environment. Usually, E2E testing is done in both the CI and the staging environment(s). Before running the E2E tests, software needs to be deployed to the test environment.

If we consider the first feature above, implementing the E2E test steps can be done so that the steps in the `Given` part of the scenario are implemented using externalized configuration. If our software system runs in a Kubernetes cluster, we can configure the microservices by creating the needed ConfigMaps. The southbound interface simulator can be controlled by launching a Kubernetes Job or, if the southbound interface simulator is a microservice with an API, commanding it via its API. After waiting for all the ingested data to be aggregated and visualized, the E2E test can launch a test tool suited for web UI testing (like TestCafe) to export chart data from the web UI to downloaded files. Then the E2E test compares the content of those files with expected values.

You can run E2E tests in a CI environment after each commit to the main branch (i.e., after a microservice CI/CD pipeline run is finished) to test that a new commit did not break any E2E tests. Alternatively, if the E2E tests are complex and take a long time to execute, you can run the E2E tests in the CI environment on a schedule, like hourly.

You can run E2E tests in a staging environment using a separate pipeline in your CI/CD tool.

Non-Functional Testing Principle

> *In addition to multi-level functional testing, non-functional testing, as automated as possible, should be performed for a software system.*

The most important categories of non-functional testing are the following:

- Performance testing
- Data volume testing
- Stability testing
- Reliability testing
- Stress and scalability testing
- Security testing

Performance Testing

The goal of performance testing is to verify the performance of a software system. This verification can be done on different levels and in different ways, for example, by verifying each performance-critical microservice separately.

To measure the performance of a microservice, performance tests can be created to benchmark the busy loop or loops in the microservice. If we take the data exporter microservice as an example, there is a busy loop that performs message decoding, transformation, and encoding. We can create a performance test using a unit testing framework for this busy loop. The performance test should execute the code in the busy loop for a certain number of rounds and verify that the execution duration does not exceed a specified threshold value obtained on the first run of the performance test. The performance test aims to verify that performance remains at the same level as it has been. If the performance has worsened, the test won't pass. In this way, you cannot accidentally introduce changes that negatively affect the performance without noticing it. This same performance test can also be used to measure the effects of optimizations. First, you write code for the busy loop without optimizations, measure the performance, and use that measure as a reference point. After that, you start introducing optimizations one by one and see if and how they affect performance.

The performance test's execution time threshold value must be separately specified for each developer's computer. This can be achieved by having a different threshold value for each computer hostname running the test.

You can also run the performance test in a CI/CD pipeline, but you must first measure the performance in that pipeline and set the threshold value accordingly. Also, the computing instances running CI/CD pipelines must be homogeneous. Otherwise, you will get different results on different CI/CD pipeline runs.

The above-described performance test was for a unit (one public function), but performance testing can also be done on the software component level. This is useful if the software component has external dependencies whose performance needs to be measured. In the telecom network analytics software system, we could introduce a performance test for the *data-ingester-service* to measure how long it takes to process a certain number of messages, like one million. After executing that test, we have a performance measurement available for reference. When we try to optimize the microservice, we can measure the performance of the optimized microservice and compare it to the reference value. If we make a change known to worsen the performance, we have a reference value to which we can compare the deteriorated performance and see if it is acceptable. And, of course, this reference value will prevent a developer from accidentally making a change that negatively impacts the microservice's performance.

We can also measure end-to-end performance. In the telecom network analytics software system, we could measure the performance from data ingestion to data export, for example.

Data Volume Testing

The goal of data volume testing is to measure the performance of a database compared when the database is empty to when the database has a sizeable amount of data stored in it. With data

volume testing, we can measure the impact of data volume on a software component's performance. Usually, an empty database has better performance than a database containing a high amount of data. This, of course, depends on the database and how it scales with a large amount of data.

Stability Testing

Stability testing aims to verify that a software system remains stable when running for an extended period of time under load. This testing is also called load, endurance, or soak testing. The term "extended period" can be interpreted differently depending on the software system. But this period should be many hours, preferably several days, even up to one week. Stability testing aims to discover problems like sporadic bugs or memory leaks. A sporadic bug is a bug that occurs only in certain conditions or at irregular intervals. A memory leak can be so small that it requires the software component to run for tens of hours after it becomes visible. It is recommended that when running the software system for a longer period, the induced load to the software system follows a natural pattern (mimicking the production load), meaning that there are peaks and lows in the load.

Stability testing can be partly automated. The load to the system can be generated using tools created for that purpose, like Apache JMeter, for example. Each software component can measure crash count, and those statistics can be analyzed automatically or manually after the stability testing is completed. Analyzing memory leaks can be trickier, but crashes due to out-of-memory should be registered, and situations where a software component is scaling out due to lack of memory.

Reliability Testing

Reliability testing aims to verify that a software system runs reliably. The software system is reliable when it is resilient to failures and recovers from failures automatically as fast as possible. Reliability testing is also called availability, recovery, or resilience testing.

Reliability testing involves chaos engineering to induce various failures in the software system’s environment. It should also ensure that the software system stays available and can automatically recover from failures.

Suppose you have a software system deployed to a Kubernetes cluster. You can make stateless services highly available by configuring them to run more than one pod. If one node goes down, it will terminate one of the pods, but the service remains available and usable because at least one other pod is still running on a different node. Also, after a short while, when Kubernetes notices that one pod is missing, it will create a new pod on a new node, and there will be the original number of pods running, and the recovery from the node down is successful.

Many parts of the reliability testing can be automated. You can use ready-made chaos engineering tools or create your own tools. Use a tool to induce failures in the environment. Then verify that services remain either highly available or at least swiftly recover from failures.

Considering the telecom network analytics software system, we could introduce a test case where the message broker (e.g., Kafka) is shut down. Then we expect alerts triggered after a while by the microservices that try to use the unavailable message broker. After the message broker is started, the alerts should cancel automatically, and the microservices should continue normal operation.

Stress and Scalability Testing

Stress testing aims to verify that a software system runs under high load. In stress testing, the software system is exposed to a load higher than the system's usual load. The software system should be designed as scalable, which means that the software system should also run under high load. Thus, stress testing should test the scalability of the software system and see that microservices scale out when needed. At the end of stress testing, the load is returned back to the normal level, and scaling the microservices in can also be verified.

You can specify a HorizontalPodAutoscaler (HPA) for a Kubernetes Deployment. In the HPA manifest, you must specify the minimum number of replicas. This should be at least two if you want to make your microservice highly available. You also need to specify the maximum number of replicas so that your microservice does not consume too many computing resources in some weird failure case. You can make the horizontal scaling (scaling in and out) happen by specifying a target utilization rate for CPU and memory. Below is an example Helm chart template for defining a Kubernetes HPA:

```
{{- if eq .Values.nodeEnv "production" }}
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "microservice.fullname" . }}
  labels:
    {{- include "microservice.labels" . | nindent 4 }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ include "microservice.fullname" . }}
  minReplicas: {{ .Values.hpa.minReplicas }}
  maxReplicas: {{ .Values.hpa.maxReplicas }}
  metrics:
    {{- if .Values.hpa.targetCPUUtilizationPercentage }}
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: {{ .Values.hpa.targetCPUUtilizationPercentage }}
    {{- end }}
    {{- if .Values.hpa.targetMemoryUtilizationPercentage }}
    - type: Resource
      resource:
```

```
        name: memory
        targetAverageUtilization: {{ .Values.hpa.targetMemoryUtilizationPercentage }}
    {{- end }}
{{- end }}
```

It is also possible to specify the autoscaling to use an external metric. An external metric could be Kafka consumer lag, for instance. If Kafka consumer lag grows too high, the HPA can scale the microservice out to have more processing power for the Kafka consumer group and when the Kafka consumer lag decreases below a defined threshold, HPA can scale the microservice in to reduce the number of pods.

Security Testing

The goal of security testing is to verify that a software system is secure and does not contain security vulnerabilities. One part of security testing is performing vulnerability scans of the software artifacts. Typically, this means scanning the microservice containers using an automatic vulnerability scanning tool. Another essential part of security testing is penetration testing, which simulates attacks by a malicious party. Penetration testing can be performed using an automated tool like OWASP ZAP or Burp Suite.

Penetration testing tools try to find security vulnerabilities in the following categories:

- Cross-site scripting
- SQL injection
- Path disclosure
- Denial of service
- Code execution
- Memory corruption
- Cross-site request forgery (CSRF)
- Information disclosure
- Local/remote file inclusion

A complete list of possible security vulnerabilities found by the OWASP ZAP tool can be found at https://www.zaproxy.org/docs/alerts/.

Other Non-Functional Testing

Other non-functional testing is documentation testing and several UI-related non-functional testing, including accessibility (A11Y) testing, visual testing, usability testing, and localization and internationalization (I18N) testing.

Visual Testing

I want to bring up visual testing here because it is important. *Backstop.js* and *cypress-plugin-snapshots* test web UI's HTML and CSS using snapshot testing. Snapshots are screenshots taken of the web UI. Snapshots are compared to ensure that the visual look of the application stays the same and there are no bugs introduced with HTML or CSS changes.

Security First Principle

Shift security implementation to the left. Implement security-related features first, not last.

Suppose that security-related features are implemented only in a very late phase of a project. In that case, there is a greater possibility of not finding time to implement them or forgetting to implement them. For that reason, security-related features should be implemented first, not last. The threat modeling process described in the next section should be used to identify the potential threats and provide a list of security features that need to be implemented as threat countermeasures.

Threat Modelling

The threat modeling process enables you to identify, quantify, and address security risks associated with an application. The threat modeling process is composed of three high-level steps:

- Decompose application
- Determine and rank threats
- Determine countermeasures and mitigation

Decompose Application

The application decomposition step is to gain knowledge of what parts the application is composed of, the external dependencies, and how they are used. This step can be performed after the application architecture is designed. The results of this step are:

- Identify an attacker's entry points to the application
- Identify assets under threat. These assets are something that an attacker is interested in
- Identify trust levels, e.g., what users with different user roles can do

Determine and Rank Threats

To determine possible threats, a threat categorization methodology should be used. The STRIDE method categorizes threats to the following categories:

Category	Description
Spoofing	Attacker acting as another user without real authentication or using stolen credentials
Tampering	Attacker maliciously changing data
Repudiation	Attacker being able to perform prohibited operations
Information disclosure	Attacker gaining access to sensitive data
Denial of service	Attacker trying to make the service unusable
Elevation of privilege	Attacker gaining unwanted access rights

The Application Security Frame (ASF) categorizes application security features into the following categories:

Category	Description
Audit & Logging	Logging user actions to detect, e.g., repudiation attacks
Authentication	Prohibit identity spoofing attacks
Authorization	Prohibit elevation of privilege attacks
Configuration Management	Proper storage of secrets and configuring the system with the least privileges
Data protection in transit and rest	Using secure protocols like TLS, encrypting sensitive information like PII in databases
Data validation	Validate input data from users to prevent, e.g., injection and ReDoS attacks
Exception management	Do not reveal implementation details in error messages to end-users

When using either of the above-described threat categorization methodologies, threats in each category should be listed based on the information about the decomposed application: what are the application entry points and assets that need to be secured? After listing potential threats in each category, the threats should be ranked. There are several ways to rank threats. The simplest way to rank threats is to put them in one of the three categories based on the risk: high, medium, and low. As a basis for the ranking, you can use information about the threat's probability and how big an adverse effect (impact) it has. The idea of ranking is to prioritize security features. Security features for high-risk threats should be implemented first.

Determine Countermeasures and Mitigation

Determining countermeasures step should provide a list of user stories for the needed security features. These security features should eliminate or at least mitigate the threats. If you have a threat that cannot be eliminated or mitigated, you can accept the risk if the threat is categorized as a low-risk threat. A low-risk threat is a threat with a low impact on the application, and the probability of the threat realization is low. Suppose you have found a threat with a very high risk in your application, and you cannot eliminate or mitigate that threat. In that case, you should eliminate the threat by completely removing the threat-related features from the application.

Security Features

Authentication and Authorization

When implementing user authentication and authorization for an application, use a 3rd party authorization service. Don't try to build an authorization service by yourself. You can easily make mistakes. Also, it can be a security risk if your application handles plaint-text user credentials. It is better to use a battle-tested solution that has the most significant bugs corrected and can store user credentials securely. We use *Keycloak* as an authorization service in the coming examples.

Also, try using established 3rd party libraries as much as possible instead of writing all authorization-related code yourself. It is also helpful to create a single frontend authentication/ authorization library and use that same library in multiple projects instead of constantly implementing authentication and authorization-related functionality from scratch in different projects.

OpenID Connect Authentication and Authorization in Frontend

Regarding frontend authorization, attention must be paid to the secure storage of authorization-related secrets like *code verifier* and *tokens*. Those must be stored in a secure location in the

browser. Below is a list of some insecure storing mechanisms:

- Cookies
    - Sent automatically, a CSRF threat
- Session/Local Storage
    - Easily stolen by malicious code (XSS threat)
- Encrypted session/local storage
    - Easily stolen by malicious code because the encryption key is in plain text
- Global variable
    - Easily stolen by malicious code (XSS threat)

Storing secrets in closure variables is not inherently insecure, but secrets are lost on page refresh or new page.

Below is an example that uses a service worker as a secure storage of secrets. The additional benefit of a service worker is that it does not allow malicious 3rd party code to modify the service worker's `fetch` method so that it can, for example, steal access tokens.

It is easy for malicious code to change the global `fetch` function:

```
fetch = () => console.log('hacked');
fetch() // prints 'hacked' to console
```

Below is a more realistic example:

```
originalFetch = fetch;
fetch = (url, options) => {
  // Implement malicious attack here
  // For example: change some data in the request body

  // Then call original fetch implementation
  return originalFetch(url, options);
}
```

Of course, one can ask: why is it possible to modify the built-in method on the global object like that? Of course, it should not be possible, but unfortunately, it is.

Let's create a Vue.js application that performs authentication and authorization using the OpenID Connect protocol, an extension of the OAuth2 protocol.

In the main module below, we set up the global `fetch` to always return an error and only allow our `tryMakeHttpRequest` function to use the original global `fetch` method. Then we register a service worker. If the service worker has already been registered, it is not registered again. Finally, we create the application (`App` component), activate the router, activate the *Pinia* middleware for state management, and mount the application to a DOM node:

main.ts

```
import { setupFetch } from "@/tryMakeHttpRequest";
setupFetch();
import { createApp } from "vue";
import { createPinia } from "pinia";
import App from "@/App.vue";
import router from "@/router";

if ("serviceWorker" in navigator) {
  await navigator.serviceWorker.register("/serviceWorker.js");
}

const app = createApp(App);
const pinia = createPinia();
app.use(pinia);
app.use(router);
app.mount("#app");
```

Below is the definition of the `App` component. After mounting, it will check whether the user is already authorized.

If the user is authorized, his/hers authorization information will be fetched from the service worker, and the user's first name will be updated in the authorization information store. The user will be forwarded to the *Home* page.

If the user is not authorized, authorization will be performed.

App.vue

```
<template>
  <HeaderView />
  <router-view></router-view>
</template>

<script setup>
import { onMounted } from "vue";
import { useRouter } from "vue-router";
import authorizationService from "@/authService";
import { useAuthInfoStore } from "@/stores/authInfoStore";
import HeaderView from "@/HeaderView.vue";
import tryMakeHttpRequest from "@/tryMakeHttpRequest";

const router = useRouter();
const route = useRoute();

onMounted(async () => {
  const response = await tryMakeHttpRequest("/authorizedUserInfo");
  const responseBody = await response.text();
  if (responseBody !== "") {
    const authorizedUserInfo = JSON.parse(responseBody);
    const { setFirstName } = useAuthInfoStore();
    setFirstName(authorizedUserInfo.firstName);
    router.push({ name: "home" });
  } else if (route.path !== '/auth') {
    authorizationService
      .tryAuthorize()
      .catch(() =>  router.push({ name: "auth-error" }));
  }
```

```
});
</script>
```

authInfoStore.ts

```
import { ref } from "vue";
import { defineStore } from "pinia";

export const useAuthInfoStore =
  defineStore("authInfoStore", () => {
    const firstName = ref('');

    function setFirstName(newFirstName: string) {
      firstName.value = newFirstName;
    }

    return { firstName, setFirstName };
  });
```

The header of the application displays the first name of the logged-in user and a button for logging the user out:

HeaderView.vue

```
<template>
  <span>{{authInfoStore.firstName}}</span>
   
  <button @click="logout">Logout</button>
</template>

<script setup>
import { useRouter } from "vue-router";
import authorizationService from "@/authService";
import { useAuthInfoStore } from "@/stores/authInfoStore";

const authInfoStore = useAuthInfoStore();
const router = useRouter();

function logout() {
  authorizationService
    .tryLogout()
    .catch(() => router.push({ name: "auth-error" }));
}
</script>
```

The `tryMakeHttpRequest` function is a wrapper around the browser's global `fetch` method. It will start an authorization procedure if an HTTP request returns the HTTP status code 403 *Forbidden*.

tryMakeHttpRequest.ts

```
import authorizationService from "@/authService";

let originalFetch: typeof fetch;

export default function tryMakeHttpRequest(
  url: RequestInfo,
  options?: RequestInit
): Promise<Response> {
```

```
  return originalFetch(url, options).then(async (response) => {
    if (response.status === 403) {
      try {
        await authorizationService.tryAuthorize();
      } catch {
        // Handle auth error, return response with status 403
      }
    }

    return response;
  });
}

export function setupFetch() {
  originalFetch = fetch;
  // @ts-ignore
  // eslint-disable-next-line no-global-assign
  fetch = () =>
    Promise.reject(new Error('Global fetch not implemented'));
}
```

Below is the implementation of the service worker:

serviceWorker.js

```
const allowedOrigins = [
  "http://localhost:8080", // IAM in dev environment
  "http://localhost:3000", // API in dev environment
  "https://software-system-x.domain.com" // prod environment
];

const apiEndpointRegex = /\/api\//;
const tokenEndpointRegex = /\/openid-connect\/token$/;
const data = {};

// Listen to messages that contain data
// to be stored inside the service worker
addEventListener("message", (event) => {
  if (event.data) {
    data[event.data.key] = event.data.value;
  }
});

function respondWithUserInfo(event) {
  const response =
    new Response(data.authorizedUserInfo
                 ? JSON.stringify(data.authorizedUserInfo)
                 : '');
  event.respondWith(response);
}

function respondWithIdToken(event) {
  const response = new Response(data.idToken
                                ? data.idToken
                                : '');
  event.respondWith(response);
}

function respondWithTokenRequest(event) {
  let body = "grant_type=authorization_code";
  body += `&code=${data.code}`;
  body += `&client_id=app-x`;
```

```
  body += `&redirect_uri=${data.redirectUri}`;
  body += `&code_verifier=${data.codeVerifier}`;
  const tokenRequest = new Request(event.request, { body });

  // Verify that state received from the authorization
  // server is same as sent by this app earlier
  if (data.state === data.receivedState) {
    event.respondWith(fetch(tokenRequest));
  } else {
    // Handle error
  }
}

function respondWithApiRequest(event) {
  const headers = new Headers(event.request.headers);

  // Add Authorization header that contains the access token
  if (data.accessToken) {
    headers.append("Authorization",
                   `Bearer ${data.accessToken}`);
  }

  const authorizedRequest = new Request(event.request, {
    headers
  });

  event.respondWith(fetch(authorizedRequest));
}

function fetchHandler(event) {
  const requestUrl = new URL(event.request.url);

  if (event.request.url.endsWith('/authorizedUserInfo') &&
      !apiEndpointRegex.test(requestUrl.pathname)) {
    respondWithUserInfo(event);
  } else if (event.request.url.endsWith('/idToken') &&
             !apiEndpointRegex.test(requestUrl.pathname)) {
    respondWithIdToken(event);
  } else if (allowedOrigins.includes(requestUrl.origin)) {
    if (tokenEndpointRegex.test(requestUrl.pathname)) {
      respondWithTokenRequest(event);
    } else if (apiEndpointRegex.test(requestUrl.pathname)) {
      respondWithApiRequest(event);
    }
  } else {
    event.respondWith(fetch(event.request));
  }
}

// Intercept all fetch requests and handle
// them with 'fetchHandler'
addEventListener("fetch", fetchHandler);
```

Authorization using the OAuth2 Authorization Code Flow is started with a browser redirect to a URL of the following kind:

```
https://authorization-server.com/auth?
response_type=code&client_id=CLIENT_ID&redirect_uri=https://example-app.com/
cb&scope=photos&state=1234zyx...ghvx3&code_challenge=CODE_CHALLENGE&code_challenge_method=SHA2
56
```

The query parameters in the above URL are the following:

- *response_type=code* - Indicates that you expect to receive an authorization code
- *client_id* - The client id you used when you created the client on the authorization server
- *redirect_uri* - Indicates the URI to redirect the browser to after authorization is completed. You need to define this URI also in the authorization server.
- *scope* - One or more scope values indicating which parts of the user's account you wish to access. Scopes should be separated by URL-encoded space characters
- *state* - A random string generated by your application, which you'll verify later
- *code_challenge* - PKCE extension: URL-safe base64-encoded SHA256 hash of the code verifier. A code verifier is a random string secret you generate
- *code_challenge_method=S256* - PKCE extension: indicates which hashing method is used (S256 means SHA256)

We need to use the PKCE extension as an additional security measure because we perform the Authorization Code Flow in the frontend instead of the backend.

If authorization is successful, the authorization server will redirect the browser to the above-given *redirect_uri* with *code* and *state* given as URL query parameters, for example:

```
https://example-app.com/cb?code=AUTH_CODE_HERE&state=1234zyx...ghvx3
```

- *code* - The authorization code returned from the authorization server
- *state* - The same state value that you passed earlier

After the application is successfully authorized, tokens can be requested with the following kind of HTTP POST request:

```
POST https://authorization-server.com/token

  grant_type=authorization_code&
  code=AUTH_CODE_HERE&
  redirect_uri=REDIRECT_URI&
  client_id=CLIENT_ID&
  code_verifier=CODE_VERIFIER
```

- *grant_type*=authorization_code - The grant type for this flow is *authorization_code*
- *code=AUTH_CODE_HERE* - This is the code you received when the browser was redirected back to your application from the authorization server.
- *redirect_uri=REDIRECT_URI* - Must be identical to the redirect URI provided earlier during authorization
- *client_id=CLIENT_ID* - The client id you used when you created the client on the authorization server

• *code_verifier=CODE_VERIFIER* - The random string secret you generated earlier

Below is the implementation of an `AuhtorizationService` class. It provides methods for authorization, getting tokens and logout.

AuthorizationService.ts

```
import pkceChallenge from "pkce-challenge";
import jwt_decode from "jwt-decode";
import tryMakeHttpRequest from "@/tryMakeHttpRequest";
import type { useAuthInfoStore } from "@/stores/authInfoStore";

interface AuthorizedUserInfo {
  readonly userName: string;
  readonly firstName: string;
  readonly lastName: string;
  readonly email: string;
}

export default class AuthorizationService {
  constructor(
    private readonly oidcConfigurationEndpoint: string,
    private readonly clientId: string,
    private readonly authRedirectUrl: string,
    private readonly loginPageUrl: string
  ) {}

  // Try to authorize the user using the OpenID Connect
  // Authorization Code Flow
  async tryAuthorize(): Promise<void> {
    // Store the redirect URI in service worker
    navigator.serviceWorker?.controller?.postMessage({
      key: "redirectUri",
      value: this.authRedirectUrl
    });

    // Store the state secret in service worker
    const state = crypto.randomUUID();
    navigator.serviceWorker?.controller?.postMessage({
      key: "state",
      value: state,
    });

    // Generate a PKCE challenge and store
    // the code verifier in service worker
    const challenge = pkceChallenge(128);
    navigator.serviceWorker?.controller?.postMessage({
      key: "codeVerifier",
      value: challenge.code_verifier,
    });

    const authUrl = await this.tryCreateAuthUrl(state, challenge);

    // Redirect the browser to authorization server's
    // authorization URL
    location.href = authUrl;
  }

  // Try get access, refresh and ID token from
  // the authorization server's token endpoint
  async tryGetTokens(
```

```
  authInfoStore: ReturnType<typeof useAuthInfoStore>
): Promise<void> {
  const oidcConfiguration = await this.getOidcConfiguration();

  const response =
    await tryMakeHttpRequest(oidcConfiguration.token_endpoint, {
      method: "post",
      mode: "cors",
      headers: {
        "Content-Type": "application/x-www-form-urlencoded",
      },
  });

  const tokens = await response.json();
  this.storeTokens(tokens);
  this.storeAuthorizedUserInfo(tokens.id_token, authInfoStore);
}

// Logout and redirect to login page
async tryLogout(): Promise<void> {
  const oidcConfiguration = await this.getOidcConfiguration();

  // Clear authorized user info in service worker
  navigator.serviceWorker?.controller?.postMessage({
    key: "authorizedUserInfo",
    value: undefined
  });

  // Get ID token from service worker
  const response = await tryMakeHttpRequest("/idToken");
  const idToken = await response.text();

  // Redirect browser to authorization server's
  // logout endpoint
  if (idToken !== "") {
    location.href =
      oidcConfiguration.end_session_endpoint +
      `?post_logout_redirect_uri=${this.loginPageUrl}` +
      `&id_token_hint=${idToken}`;
  } else {
    location.href = oidcConfiguration.end_session_endpoint;
  }
}

private async getOidcConfiguration(): Promise<any> {
  const response =
    await tryMakeHttpRequest(this.oidcConfigurationEndpoint);

  return response.json();
}

private async tryCreateAuthUrl(
  state: string,
  challenge: ReturnType<typeof pkceChallenge>
) {
  const oidcConfiguration = await this.getOidcConfiguration();
  let authUrl = oidcConfiguration.authorization_endpoint;

  authUrl += "?response_type=code";
  authUrl += "&scope=openid+profile+email";
  authUrl += `&client_id=${this.clientId}`;
  authUrl += `&redirect_uri=${this.authRedirectUrl}`;
  authUrl += `&state=${state}`;
```

```
    authUrl += `&code_challenge=${challenge.code_challenge}`;
    authUrl += "&code_challenge_method=S256";

    return authUrl;
  }

  private storeTokens(tokens: any) {
    navigator.serviceWorker?.controller?.postMessage({
      key: "accessToken",
      value: tokens.access_token,
    });

    navigator.serviceWorker?.controller?.postMessage({
      key: "refreshToken",
      value: tokens.refresh_token,
    });

    navigator.serviceWorker?.controller?.postMessage({
      key: "idToken",
      value: tokens.id_token,
    });
  }

  private storeAuthorizedUserInfo(
    idToken: any,
    authInfoStore: ReturnType<typeof useAuthInfoStore>
    ) {
    const idTokenClaims: any = jwt_decode(idToken);

    const authorizedUserInfo = {
      userName: idTokenClaims.preferred_username,
      firstName: idTokenClaims.given_name,
      lastName: idTokenClaims.family_name,
      email: idTokenClaims.email,
    };

    navigator.serviceWorker?.controller?.postMessage({
      key: "authorizedUserInfo",
      value: authorizedUserInfo
    });

    authInfoStore.setFirstName(idTokenClaims.given_name);
  }
}
```

Below is an example response you get when you execute the `tryMakeHttpRequest` function in the `tryGetTokens` method:

```
{
  "access_token": "eyJz93a...k4laUWw",
  "id_token": "UFn43f...c5vvfGF",
  "refresh_token": "GEbRxBN...edjnXbL",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

The `AuthorizationCallback` component is the component that will be rendered when the authorization server redirects the browser back to the application after successful authorization. This component stores the authorization code and the received state in the service worker and

initiates a request for tokens. After receiving tokens, it will route the application to the home page. As an additional security measure, the token request will only be performed if the original *state* and *received state* are equal. This check is done in the service worker code.

AuthorizationCallback.vue

```
<template>
  <div></div>
</template>

<script setup>
import { onMounted } from "vue";
import { useRouter, useRoute } from "vue-router";
import authorizationService from "@/authService";
import { useAuthInfoStore } from "@/stores/authInfoStore";

const { query } = useRoute();
const router = useRouter();
const authInfoStore = useAuthInfoStore();

onMounted(async () => {
  // Store authorization code in service worker
  navigator.serviceWorker?.controller?.postMessage({
    key: "code",
    value: query.code,
  });

  // Store received state in service worker
  navigator.serviceWorker?.controller?.postMessage({
    key: "receivedState",
    value: query.state,
  });

  // Try fetch tokens
  try {
    await authorizationService.tryGetTokens(authInfoStore);
    router.push({ name: "home" });
  } catch (error) {
    router.push({ name: "auth-error" });
  }
});
</script>
```

Other UI components the application uses are defined below:

AuthorizationError.vue

```
<template>
  <div>Error</div>
</template>
```

LoginView.vue

```
<template>
  <div>Login</div>
</template>
```

HomeView.vue

```
<template>
  <div>Home</div>
</template>
```

The application's router is the following:

router.ts

```
import { createRouter, createWebHistory } from "vue-router";
import AuthorizationCallback from "@/AuthorizationCallback.vue";
import AuthorizationError from "@/AuthorizationError.vue";
import HomeView from "@/HomeView.vue";
import LoginView from "@/LoginView.vue";

const routes = [
  {
    path: "/",
    name: "login",
    component: LoginView,
  },
  {
    path: "/auth",
    name: "auth",
    component: AuthorizationCallback,
  },
  {
    path: "/auth-error",
    name: "auth-error",
    component: AuthorizationError,
  },
  {
    path: "/home",
    name: "home",
    component: HomeView,
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

The below *authService* module contains definitions of needed constants and creates an instance of the `AuthorizationService` class. The below code contains values for a local development environment. In real life, these values should be taken from environment variables. The below values work if you have a Keycloak service running at *localhost:8080* and the Vue app running at *localhost:5173*. You must create a client in the Keycloak with the name 'app-x'. Additionally, you must define a valid redirect URI and add an allowed web origin. Lastly, you must configure a valid post-logout redirect URI (see the below image). The default access token lifetime in Keycloak is just one minute. You can increase that for testing purposes in the realm settings (the token tab)

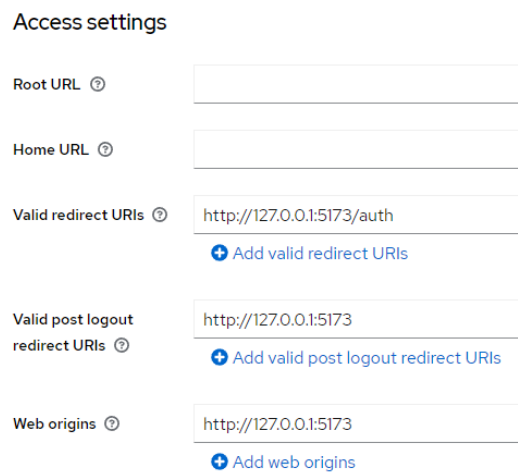

Fig 6.1 Keycloak Settings for the Client

authService.ts

```
import AuthorizationService from "@/AuthorizationService";

const oidcConfigurationEndpoint =
  "http://localhost:8080/realms/master/.well-known/openid-configuration";

const clientId = "app-x";
const redirectUrl = "http://127.0.0.1:5173/auth";
const loginPageUrl = "http://127.0.0.1:5173";

const authorizationService = new AuthorizationService(
  oidcConfigurationEndpoint,
  clientId,
  redirectUrl,
  loginPageUrl
);

export default authorizationService;
```

OAuth2 Authorization in Backend

Only let authorized users access resources. The best way not to forget to implement authorization is to deny access to resources by default. You can require that an authorization annotation must be present in all controller methods. If an API endpoint does not require authorization, a special annotation like `@AllowAnyone` could be used. If a controller method is missing an authorization annotation, an exception can be thrown, for example. This way, you can never forget to add an authorization annotation to a controller method.

Broken access control is number one in the OWASP Top 10 for 2021. Especially remember to disallow users to create, view, edit or delete resources belonging to someone else by providing someone else's unique identifier.

Below is a JWT-based authorizer class that can be used in a backend API service implemented with the Express framework:

JwtAuthorizer.ts

```
import _ from 'lodash';
import { decode, verify } from 'jsonwebtoken';
import { fetch } from 'fetch-h2';
import { Request } from 'express';
import jwks from 'jwks-rsa';
import throwException from './throwException';

export default class JwtAuthorizer implements
                      Authorizer {
  private readonly oidcConfigurationEndpoint: string;
  private readonly rolesClaimPath: string;
  private readonly getUserIdBySubjectUrl: string;
  private jwksClient: any;

  constructor() {
    this.oidcConfigurationEndpoint =
     process.env.OIDC_CONFIGURATION_ENDPOINT ??
      throwException('OIDC_CONFIGURATION_ENDPOINT is not defined');

    // With Keycloak you can use e.g., realm_access.roles
    this.rolesClaimPath = process.env.JWT_ROLES_CLAIM_PATH ??
      throwException('JWT_ROLES_CLAIM_PATH is not defined')

    // This is the URL where you can fetch the user id for a
    // specific 'sub' claim value in the access token
    this.getUserIdBySubjectUrl =
      process.env.GET_USER_ID_BY_SUBJECT_URL ??
      throwException('GET_USER_ID_BY_SUBJECT_URL is not defined');
  }

  async tryAuthorize(
    request: Request
  ): Promise<void> {
    await this.tryGetClaims(request.headers.authorization);
  }

  // Authorize a user for his/hers own resources only
  // Note! For some IAM systems other than Keycloak,
  // you might need to use 'uid'
  // claim instead of 'sub' to get really unique user id
  async tryAuthorizeForUserOwnResourcesOnly(
    userId: number,
    request: Request
  ): Promise<void> {
    const claims =
      await this.tryGetClaims(request.headers.authorization);

    const response =
      await fetch(this.getUserIdBySubjectUrl +
                  `?sub=${claims.sub}`);

    const userIdObject = await response.json();

    if (userId !== userIdObject.userId) {
      throw new Error('...');
    }
  }
```

```
  async tryAuthorizeIfUserHasOneOfRoles(
    allowedRoles: string[],
    request: Request):
  Promise<void> {
    const claims =
      await this.tryGetClaims(request.headers.authorization);

    const roles = _.get(claims, this.rolesClaimPath);

    const isAuthorized =
      allowedRoles.some((allowedRole) =>
        roles.includes(allowedRole));

    if (!isAuthorized) {
      throw new Error('...');
    }
  }

  private async tryGetClaims(
    authHeader: string | undefined
  ): Promise<any> {
    const response = await fetch(this.oidcConfigurationEndpoint);
    const oidcConfiguration = await response.json();

    if (!this.jwksClient) {
      this.jwksClient =
        jwks({ jwksUri: oidcConfiguration.jwks_uri });
    }

    const jwt = authHeader?.split('Bearer ').pop();
    const decodedJwt = decode(jwt ?? '', { complete: true });
    const kid = decodedJwt?.header?.kid;
    const signingKey = await this.jwksClient.getSigningKey(kid);
    return verify(jwt ?? '', signingKey.getPublicKey());
  }
}

export const authorizer = new JwtAuthorizer();
```

Below is an example API service that utilizes the above-defined `JwtAuthorizer`:

app.ts

```
import express, { Response, Request } from 'express';
import cors from 'cors';
import { authorizer } from './JwtAuthorizer';

const app = express();
app.use(express.json());
app.use(cors());

function sendAuthError(request: Request, response: Response) {
  if (request.headers.authorization) {
    response
      .status(403)
      .json({ error: 'Unauthorized' });
  } else {
    response
      .status(401)
      .json({ error: 'Unauthenticated' });
  }
```

```
}

app.get('/api/sales-items', () => {
  // No authorized needed
  // Send sales items
});

app.post('/api/messages', (request, response) => {
  authorizer.tryAuthorize(request)
    .then(() => {
      // Create a message
    }).catch(() => sendAuthError(request, response));
});

app.post('/api/orders', (request, response) => {
  authorizer
    .tryAuthorizeForUserOwnResourcesOnly(request.body.userId,
                                         request)
    .then(() => {
      // Create an order for the user,
      // user cannot create orders for other users
    }).catch(() => sendAuthError(request, response));
});

app.delete('/api/sales-items', (request, response) => {
  authorizer
    .tryAuthorizeIfUserHasOneOfRoles(['admin'], request)
    .then(() => {
      // Only admin user can delete all sales items
    }).catch(() => sendAuthError(request, response));
});

app.listen(3000);
```

Below is a simple example of implementing an OAuth2 resource server using Spring Boot. First, you need to add the OAuth2 resource server dependency:

build.gradle

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
}
```

Next, you need to configure a JWT issuer URL. In the below example, we use a localhost Keycloak service.

application.yml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/realms/master/
```

Then you will configure which API endpoints need to be authorized with a valid JWT. The below example requires requests to all API endpoints to contain a valid JWT.

SecurityConfiguration.java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
public class SecurityConfiguration
                extends WebSecurityConfigurerAdapter {
  @Override
  protected final void configure(
    final HttpSecurity httpSecurity
  ) throws Exception {
    httpSecurity.authorizeRequests()
      .antMatchers("/**")
      .authenticated()
      .and()
      .oauth2ResourceServer()
      .jwt();
  }
}
```

In a REST controller, you can inject the JWT using the `AuthenticationPrincipal` annotation to perform additional authorization in the controller methods:

SalesItemController.java

```
@RestController
@RequestMapping(SalesItemController.API_ENDPOINT)
@Slf4j
public class SalesItemController {
  public static final String API_ENDPOINT = "/salesitems";

  @Autowired
  private SalesItemService salesItemService;

  @PostMapping
  @ResponseStatus(HttpStatus.CREATED)
  public final SalesItem createSalesItem(
    @RequestBody final SalesItemArg salesItemArg,
    @AuthenticationPrincipal final Jwt jwt
  ) {
    log.info("Username: {}",
             jwt.getClaimAsString("preferred_username"));

    // You can now use the "jwt" object to get a claim about
    // user's roles and verify a needed role, for example.
    return salesItemService.createSalesItem(salesItemArg);
  }
}
```

Password Policy

Implement a password policy requiring strong passwords and prefer passphrases over passwords. A passphrase is supposed to contain multiple words. Passphrases are harder to guess by attackers and

easier to remember by users than strong passwords. Allow passphrases to contain Unicode characters. This allows users to create passphrases using their mother tongue.

You should require that passwords are strong and match the following criteria:

- At least 12 characters long
- At least one uppercase letter
- At least one lowercase letter
- At least one number
- At least one special character
- May not contain the username
- May not contain too many identical digits or letters, e.g., a password containing "111111", "aaaaaa," or "1a1a1a1a1a" should be denied
- May not contain too many consecutive numbers or letters, e.g., a password containing "12345", "56789", "abcdef", or "klmno" should be denied
- May not contain too many adjacent letters in the keyboard, e.g., a password containing "qwerty" should be denied
- May not contain a black-listed word. Black-list all commonly used, easy-to-guess passwords.

Cryptography

The following are the key security features to implement related to cryptography:

- Do not transmit data in clear text
    - You don't need to implement HTTPS in all the microservices because you can set up a service mesh and configure it to implement mTLS between services
- Do not store sensitive information like personally identifiable information (PII) in clear text
    - Encrypt sensitive data before storing it in a database and decrypt it upon fetching from the database
    - Remember to identify which data is classified as sensitive according to privacy laws, regulatory requirements, or business needs
    - Do not use legacy protocols such as FTP and SMTP for transporting sensitive data
    - Discard sensitive data as soon as possible or use tokenization (e.g., PCI DSS compliant) or even truncation
    - Do not cache sensitive data
- Do not use old/weak cryptographic algorithms. Use robust algorithms like SHA-256 or AES-256
- Do not allow using default passwords or encryption keys in a production environment
    - You can implement validation logic for passwords/encryption keys in microservices when the microservices run in production. If a microservice's passwords/encryption keys are not strong enough, the microservice should not run but exit with an error

Denial-of-service (DoS) Prevention

DoS prevention should happen at least in the following ways:

- Establish request rate limiting for microservices. It can be done at the API gateway level or by the cloud provider
- Use Captcha to prevent non-human (robotic) users from performing potentially expensive operations like creating new resources or fetching large resources, like large files, for example

SQL Injection Prevention

The following are the key features to implement to prevent SQL injection attacks:

- Use parameterized SQL statements. Do not concatenate user-supplied data directly to an SQL statement string
- Remember that you cannot use parameterization in all parts of an SQL statement. If you must put user-supplied data into an SQL statement without parameterization, sanitize/validate it first. For example, for LIMIT, you must validate that the user-supplied value is an integer and in a given range
- Migrate to use ORM (Object Relational Mapping)
- Use proper limiting on the number of fetched records within queries to prevent mass disclosure of records
- Verify the correct shape of the first query result row. Do not send the query result to the client if the shape of the data in the first row is wrong, e.g., it contains the wrong fields.

Security Configuration

By default, the security context for containers should be the following:

- Container should not be privileged
- All capabilities are dropped
- Container filesystem is read-only
- Only a non-root user is allowed to run inside the container
- Define the non-root user and group under which the container should run
- Disallow privilege escalation

Implement the sending of security-related HTTP response headers in the API gateway:

- `X-Content-Type-Options: nosniff`
- `Strict-Transport-Security: max-age: ; includeSubDomains`

- `X-Frame-Options: DENY`
- `Content-Security-Policy: frame-ancestors 'none'`
- `Content-Type: application/json`
- If caching is not specifically enabled and configured, the following header should be set: `Cache-Control: no-store`
- `Access-Control-Allow-Origin: https://your_domain_here`

Automatic Vulnerability Scanning

Implement automatic vulnerability scanning in microservice CI/CD pipelines and the container registry at regular intervals. All software components of the software system should be scanned. Correct at least all critical and high vulnerabilities immediately.

Integrity

Use only container images with tags that have an SHA digest. If an attacker succeeds in publishing a malicious container image with the same tag, the SHA digest prevents from taking that malicious image into use. Ensure you use libraries and dependencies from trusted sources, like NPM or Maven. You can also host internal mirrors of repositories to avoid accidentally using any untrusted repository. Ensure a review process exists for all code (source, deployment, infrastructure) and configuration changes so that no malicious code can be introduced into your software system.

Error Handling

Ensure that error messages in API responses do not contain sensitive information. Do not add stack traces to error responses transmitted to clients in a production environment.

Audit Logging

Auditable end-user-related events, such as logins, failed logins, unauthorized or invalid requests, and high-value transactions, should be logged and stored in an external audit logging system. The audit logging system should automatically detect suspicious action related to an end-user and alert about it.

Input Validation

Always validate input from untrusted sources, like from an end-user. There are many ways to implement validation, and several libraries are available for that purpose. Let's assume you use

ORM and implement entities and data transfer objects. The best way to ensure proper validation is to require that each entity/DTO property must have a validation annotation. If a property in an entity or DTO does not require any validation, annotate that property with a special annotation, like `@AnyValue`, for example.

Remember to validate unvalidated data from all possible sources:

- Command line arguments
- Environment variables
- Standard input (stdin)
- File from the file system
- Data from a socket (network input)
- UI input

Validating Numbers

When validating numeric values, always validate that a value is in a specified range. For example, if you use an unvalidated number to check if a loop should end and that number is very large, it can cause a denial of service (DoS). If a number should be an integer, don't allow floating-point values.

Validating Strings

When validating a string, always validate the maximum length of the string first. Only after that perform additional validation. Validating a long string using a regular expression can cause a regular expression denial of service (ReDoS). You should avoid crafting your own regular expressions for validation purposes, instead use a ready-made library that contains battle-tested code. Consider also using the Google RE2 library (https://github.com/google/re2). It is safer than regular expression functionality provided by many language runtimes, and your code will be less susceptible to ReDoS.

Validating Arrays

When validating an array, you can validate the size of the array not being too small or large, and you can validate the uniqueness of values if needed. Also, remember to validate each value in the array separately.

Validating Objects

Validate an object by validating each property of the object separately. Remember to validate nested objects also.

Validation Library Example

Check *Appendix A* for creating a TypeScript validation library using the *validated-types* NPM library. The validation library can validate JavaScript objects, e.g., parsed JSON/YAML-format configuration, environment variables (process.env object), and input DTOs. The validation library accepts an object schema and validates an object according to the given schema and produces a strongly typed validated object.

API Design Principles

This chapter presents design principles for both frontend facing and backend APIs. First, frontend-facing API design is discussed, and then inter-microservice API design is covered.

Frontend Facing API Design Principles

Most frontend facing APIs should be HTTP-based JSON-RPC, REST, or GraphQL APIs. Use GraphQL especially when the API handles heavily nested resources, or clients want to decide what fields queries should return. For subscription-based APIs, use Server-Sent Events (SSE) or GraphQL subscriptions, and for real-time bidirectional communication, use WebSocket.

JSON-RPC API Design Principle

> *Design a JSON-RPC API to perform a single action (procedure) for an API endpoint.*

As the name says, JSON-RPC APIs are for executing remote procedure calls. The remote procedure argument is a JSON object in the HTTP request body. And the remote procedure return value is a JSON object in the HTTP response body. A client calls a remote procedure by issuing an HTTP POST request where it specifies the name of the procedure in the URL path and gives the argument for the remote procedure call in the request body in JSON.

Below is an example request for a translation service's *translate* procedure:

```
POST /translation-service/translate
{
  "text": "Ich liebe dich"
  "fromLanguage": "German",
  "toLanguage": "English"
}
```

The API server shall respond with an HTTP status code and include the procedure's response in the HTTP response body in JSON.

For the above request, you get the following response:

```
HTTP/1.1 200 OK
{
  "translatedText": "I love you"
}
```

Let's have another example with a *web-page-search-service*:

```
POST /web-page-search-service/search-web-pages
{
  "containingText": "Software design patterns"
}
```

```
HTTP/1.1 200 OK
[
  {
    "url": "https://...",
    "title: "...",
    "date": "...",
    "contentExcerpt": "..."
  },
  ...
]
```

You can create a complete service using JSON-RPC instead of REST or GraphQL. Below are five remote procedures defined for a *sales-item-service*. The benefit of using JSON-RPC instead of REST, GraphQL, or gRPC, is that you don't have to learn any specific technology.

```
POST /sales-item-service/createSalesItem
{
  "name": "Sample sales item",
  "price": 20
}

POST /sales-item-service/getSalesItems

POST /sales-item-service/getSalesItemById
{
  "id": 1
}

POST /sales-item-service/updateSalesItem
{
  "id": 1,
  "name": "Sample sales item name modified",
  "price": 30
}

POST /sales-item-service/deleteSalesItemById
{
  "id": 1
}
```

```
POST /sales-item-service/deleteSalesItems
```

You can easily create a controller for the above service. Below is an example of such a controller with one remote procedure defined:

SalesItemController.java

```
@RestController
public class SalesItemController {
  @Autowired
  private SalesItemService salesItemService;

  @PostMapping("/createSalesItem")
  @ResponseStatus(HttpStatus.CREATED)
  public final SalesItem createSalesItem(
    @RequestBody final SalesItemArg salesItemArg
  ) {
    return salesItemService.createSalesItem(salesItemArg);
  }

  // Rest of the methods ...
}
```

You can version your API by adding a version number to the URL path. In the below example, the new API version 2 allows a new procedure argument `someNewParam` to be supplied for the `search-web-pages` procedure.

```
POST /web-page-search-service/v2/search-web-pages
{
  "containingText": "Software design patterns"
  "someNewParam": "..."
}
```

REST API Design Principle

> *Design a REST API for interaction with a resource (or resources) using CRUD (create, read, update, delete) operations.*

Many APIs fall into the category of performing CRUD operations on resources. Let's create an example REST API called *sales-item-service* for performing CRUD operations on sales items.

Creating a Resource

Creating a new resource using a REST API is done by sending an HTTP POST request to the API's resource endpoint. The API's resource endpoint should be named according to resources it handles. The resource endpoint name should be a noun and always given in the plural form, for example, for the *sales-item-service*, the resource endpoint should be *sales-items*, and for an *order-service* handling orders, the resource endpoint should be called *orders*.

You give the resource to be created in the HTTP request body in JSON. To create a new sales item, you can issue the following request:

```
POST /sales-item-service/sales-items
{
  "name": "Sample sales item",
  "price": 20
}
```

The server will respond with the HTTP status code 201 *Created*. The server can add fields to the resource upon creation. Typically, the server will add an `id` property to the created resource, but it can add other properties also. The server will respond with the created resource in the HTTP response body in JSON. Below is a response to a sales item creation request. You can notice that the server added the `id` property to the resource. Other properties that are usually added are the creation timestamp and the version of the resource (the version of a newly created resource should be one).

```
HTTP/1.1 201 Created
{
  "id": 1,
  "name": "Sample sales item",
  "price": 20
}
```

If the supplied resource to be created is somehow invalid, the server should respond with the HTTP status code 400 *Bad Request* and explain the error in the response body. The response body should be in JSON format containing information about the error, like the error code and message.

If the created resource is huge, there is no need to return the resource to the caller and waste network bandwidth. You can return the added properties only. For example, if the server only adds the `id` property, it is possible to return only the `id` in the response body:

```
HTTP/1.1 201 Created
{
  "id": 1
}
```

The request sender can construct the created resource by merging the sent resource object with the received resource object.

Ensure that no duplicate resources are created.

When a client tries to create a new resource, the resource creation request may fail so that the resource was created successfully created on the server, but the client did not receive a response on time, and the request failed due to timeout. From the server's point of view, the request was successful, but from the client's point of view, it was indeterminate. The client, of course, needs to re-issue the time-outed request, and if it succeeds, the same resource is created twice on the server side, which is probably always unwanted.

Suppose a resource contains a unique property, like a user's email. In that case, it is impossible to create a duplicate resource if the server is correctly implemented (= the unique property is marked as a unique column in the database table definition). In many cases, such a unique field does not exist in the resource. In those cases, the client can supply a universally unique identifier (UUID), named `creationUuid`, for example. The role of the server is to check if a resource with the same `creationUuid` was already created and to fail the creation of a duplicate resource. As an alternative to the UUID approach, the server can ask for verification from the client if the creation of two identical resources is intended in case the server receives two identical resources from the same client during a short period of time.

Reading Resources

Reading resources with a REST API is done by sending an HTTP GET request to the API's resource endpoint. To read all sales items, you can issue the following request:

```
GET /sales-item-service/sales-items
```

The server will respond with the HTTP status code 200 *OK*. The server will respond with a JSON array of resources in the response body or an empty array in case none is found. Below is an example response to a request to get the sales items:

```
HTTP/1.1 200 OK
[
  {
    "id": 1,
    "name": "Sample sales item",
    "price": 20
  }
]
```

To read a single resource by its id, add the resource id to the request URL path as follows:

```
GET /sales-item-service/sales-items/<id>
```

The following request can be issued to read the sales item identified with id 1:

```
GET /sales-item-service/sales-items/1
```

The server responds with the HTTP status code 404 *Not Found* if the requested resource is not found.

You can define parameters in the URL query string to filter what resources to read. A query string is the last part of the URL and is separated from the URL path by a question mark (?) character. A query string can contain one or more parameters separated by ampersand (&) characters. Each query string parameter has the following format: <*query-parameter-name*>=<*query-parameter-value*>. Below is an example request with two query parameters: *name-contains* and *price-greater-*

than.

```
GET /sales-item-service/sales-items?name-contains=Sample&price-greater-than=10
```

The above request gets sales items whose name contains the string "*Sample*" and whose price is greater than 10.

To define a filter, you can specify a query parameter in the following format: *<fieldName>[-<condition>]=<value>*, for example:

- price=10
- price-not-equal=10
- price-less-than=10
- price-less-than-equal=10
- price-greater-than=10
- price-greater-than-equal=10
- name-starts-with=Sample
- name-ends-with=item
- name-contains=Sample
- createdTimestamp-before=2022-08-02T05:18:00Z
- createdTimestamp-after=2022-08-02T05:18:00Z
- images.url-starts-with=https

Remember that when implementing the server side and adding the above-given parameters to an SQL query, you must use a parameterized SQL query to prevent SQL injection attacks because an attacker can send malicious data in the query parameters.

Other actions like projection, sorting, and pagination for the queried resources can also be defined with query parameters in the URL:

```
GET /sales-item-service/sales-items?fields=id,name&sort-by=price:asc&offset=0&limit=100
```

The above request gets sales items sorted by price (ascending). The number of fetched sales items is limited to 100. Sales items are fetched beginning from the offset 0, and the response only contains fields *id* and *name* for each sales item.

The *fields* parameter defines what resource fields (properties) are returned in the response. The wanted fields are defined as a comma-separated list of field names. If you want to define sub-resource fields, those can be defined with the dot notation, for example:

```
fields=id,name,images.url
```

The *sort-by* query parameter defines sorting using the following format *sort-by=<fieldName>:asc*|

desc,[<fieldName>:asc|desc].

For example:

```
sort-by=price:asc,images.rank:asc
```

In the above example, the resources are returned sorted first by ascending price and secondarily by image's rank.

The *limit* and *offset* parameters are used for pagination. The *limit* query parameter defines the maximum number of resources that can be returned. The *offset* query parameter specifies the offset from which resources are returned. You can also paginate sub-resources by giving the *offset* and *limit* in the form of *<sub-resource>:<number>*. Below is an example of using pagination query parameters:

```
offset=0&limit=50,images:5
```

The above query parameters define that the first page of 50 sales items is fetched, and each sales item contains the first five images of the sales item. Instead of *offset* and *limit* parameters, you can use *page* and *pageSize* parameters. The *page* parameter defines the page number, and the *pageSize* defines how many resources a page should contain.

Remember to validate user-supplied data to prevent SQL injection attacks when implementing the server side and adding data from query parameters to an SQL query. For example, field names in the *fields* query parameter should only contain characters allowed in an SQL column name. Similarly, the value of the *sort-by* parameter should only contain characters allowed in an SQL column name and words *asc* and *desc*. And finally, the values of the *offset* and *limit* (or *page* and *pageSize*) parameters must be integers. You should also validate the *limit/pageSize* parameter against the maximum allowed value because you should not allow end-users to fetch too many resources at a time.

Some HTTP servers log the URL of an HTTP GET request. For this reason, it is not recommended to put sensitive information in the URL. Sensitive information should be put into a request body. Also, browsers can have a limit for the maximum length of an URL. If you have a query string that is thousands of characters long, you should give parameters in the request body instead. You should not put a request body to an HTTP GET request. What you should do is issue the request using the HTTP POST method instead, for example:

```
POST /sales-item-service/sales-items
X-HTTP-Method-Override: GET
{
  "fields": ["name"],
  "sortBy": "price:asc",
  "limit": 100
}
```

The server can confuse the above request with a sales item creation request because the URL and

the HTTP method are identical to a resource creation request. For this reason, a custom HTTP request header *X-HTTP-Method-Override* has been added to the request. The server should read the custom header and treat the above request as a GET request. The *X-HTTP-Method-Override* header tells the server to override the request method with the method supplied in the header.

Updating Resources

Updating a resource with a REST API is done by sending an HTTP PUT or PATCH request to the API's resource endpoint. To update the sales item identified with id 1, you can issue the following request:

```
PUT /sales-item-service/sales-items/1
{
  "name": "Sample sales item name modified",
  "price": 30
}
```

The server will respond without content:

```
HTTP/1.1 204 No Content
```

The server will respond with the HTTP status code 404 *Not Found* if the requested resource is not found.

If the supplied resource in the request is invalid, the server should respond with the HTTP status code 400 *Bad Request*. The response body should contain an error object in JSON.

HTTP PUT request will replace the existing resource with the supplied resource. You can also modify an existing resource partially using the HTTP PATCH method:

```
PATCH /sales-item-service/sales-items/1
{
  "price": 30
}
```

The above request only modifies the price property of the sales item identified with id 1.

You can do bulk updates by specifying a filter in the URL, for example:

```
PATCH /sales-item-service/sales-items?price-less-than=10
{
  "price": 10
}
```

The above example will update the price property of each resource where the price is less than ten currently. On the server side, the API endpoint could use the following parameterized SQL statement to implement the update functionality:

```
UPDATE salesitems SET price = :price WHERE price < :priceLessThan
```

The above SQL statement will only modify the price column, and other columns will remain intact.

Use resource versioning when needed.

When you get a resource from the server and then try to update it, it is possible that someone else has updated it after you got it and before you try to update it. Sometimes this can be ok if you don't care about other clients' updates. But sometimes, you want to ensure no one else has updated the resource before you update it. In that case, you should use resource versioning. In the resource versioning, there is a version field in the resource, which is incremented by one during each update. If you get a resource with version x and then try to update the resource giving back the same version x to the server, but someone else has updated the resource to version x + 1, your update will fail because of the version mismatch (x != x + 1). The server should respond with the HTTP status code 409 *Conflict*. After receiving the conflict response, you can fetch the latest version of the resource from the server and, based on the resource's new state, decide whether your update is still relevant or not.

The server should assign the resource version value to the HTTP response header *ETag*. A client can use the received ETag value in a conditional HTTP GET request by assigning the received ETag value to the request header *If-None-Match*. Now the server will return the requested resource only if it has a newer version. Otherwise, the server returns nothing with the HTTP status code 304 *Not Modified*. This has the advantage of not needing to transfer an unmodified resource from the server to the client. This can be especially beneficial when the resource is large or the connection between the server and the client is slow.

Deleting Resources

Deleting a resource with a REST API is done by sending an HTTP DELETE request to the API's resource endpoint. To delete the sales item identified with id 1, you can issue the following request:

```
DELETE /sales-item-service/sales-items/1
```

The server will respond without content:

```
HTTP/1.1 204 No Content
```

If the resource requested to be deleted has already been deleted, the API should still respond with the HTTP status code 204 *No Content*, meaning a successful operation. It should not respond with the HTTP status code 404 *Not Found*.

To delete all sales items, you can issue the following request:

```
DELETE /sales-item-service/sales-items
```

To delete sales items using a filter, you can issue the following kind of request:

```
DELETE /sales-item-service/sales-items?price-less-than=10
```

On the server side, the API endpoint handler can use the following parameterized SQL query to implement the deleting functionality:

```
DELETE FROM salesitems WHERE price < ?
```

Executing Non-CRUD Actions on Resources

Sometimes you need to perform non-CRUD actions on resources. In those cases, you can issue an HTTP POST request and put the name of the action (a verb) after the resource name in the URL. The below example will perform a *deposit* action on an account resource:

```
POST /account-balance-service/accounts/12345678912/deposit
{
  "amountInCents": 2510
}
```

Similarly, you can perform a withdrawal action:

```
POST /account-balance-service/accounts/12345678912/withdraw
{
  "amountInCents": 2510
}
```

Resource Composition

A resource can be composed of other resources. There are two ways to implement resource composition: Nesting resources or linking resources. Let's have an example of nesting resources first. A sales item resource can contain one or more image resources. We don't want to return all images when a client requests a sales item because images can be large and are not necessarily used by the client. What we could return is a set of small thumbnail images. For a client to view the images of a sales item, we could implement an API endpoint for image resources. To get images for a specific sales item, the following API call can be issued:

```
GET /sales-item-service/sales-items/<id>/images
```

You can also add a new image for a sales item:

```
POST /sales-item-service/sales-items/<id>/images
```

Also, other CRUD operations could be made available:

```
PUT /sales-item-service/sales-items/<salesItemId>/images/<imageId>
DELETE /sales-item-service/sales-items/<salesItemId>/images/<imageId>
```

The problem with this approach is that the *sales-item-service* will grow in size and if you need to add more nested resources in the future, the size will grow even more, making the microservice too complex and being responsible for too many things.

A better alternative is to create a separate microservice for the nested resources. This will enable utilizing the best-suited technologies to implement a microservice. Regarding the sales item images, the *sales-item-image-service* could employ a cloud object storage to store images, and the *sales-item-service* could utilize a standard relational database for storing sales items.

When having a separate microservice for sales item images, you can get the images for a sales item by issuing the following request:

```
GET /sales-item-image-service/sales-item-images?salesItemId=<salesItemId>
```

You can notice that the *sales-item-service* and *sales-item-image-service* are now linked by the *salesItemId*.

HTTP Status Codes

HTTP Status Code	When To Use
200 OK	Successful API operations with the GET method
201 Created	Successful API operations with the POST method
204 No Content	Successful API operations with the PUT, PATCH, or DELETE method
400 Bad Request	Client error in API operations, e.g., invalid data supplied by the client
401 Unauthorized	Client does not provide an authorization header in the request
403 Forbidden	Client provides an authorization header in the request, but the user is not authorized to perform the API operation
404 Not Found	When requesting a non-existent resource with the GET, PUT, or PATCH method
405 Method Not Allowed	When a client tries to use the wrong method for an API endpoint
406 Not Acceptable	When a client requests a response in a format that the server cannot produce, e.g., requests XML, but the server provides only JSON
409 Conflict	When a client is trying to update a resource that has been updated after the client got the resource

HTTP Status Code	When To Use
413 Payload Too Large	When a client tries to supply a too large payload in a request. To prevent DoS attacks, do not accept arbitrarily large payloads from clients
429 Too Many Requests	Configure rate limiting in your API gateway to send this status code when the request rate is exceeded
500 Internal Server Error	When a server error occurs, for example, an exception is thrown
503 Service Unavailable	Server's connections to dependent services fail. This indicates that clients should retry the request after a while because this issue is usually temporary.

HATEOAS and HAL

Hypermedia as the Engine of Application State (HATEOAS) can be used to add hypermedia/metadata to a requested resource. *Hypertext Application Language* (HAL) is a convention for defining hypermedia (metadata), such as links to external resources. Below is an example response to a request that fetches the sales item with id 1234. The sales item is owned by the user with id 5678. The response provides a link to the fetched resource itself and another link to fetch the user (account) that owns the sales item:

```
{
  "_links": {
    "self": {
      "href": "https://.../sales-item-service/sales-items/1234"
    },
    "userAccount": {
      "href": "https://.../user-account-service/user-accounts/5678"
    }
  },
  "id": 1234,
  "name": "Sales item xyz"
  "userAccountId": 5678
}
```

When fetching a collection of sales items for page 3 using HAL, we can get the following kind of response:

```
{
  "_links": {
    "self": {
      "href": "https://.../sales-items?page=3"
    },
    "first": {
      "href": "https://...sales-items"
    },
    "prev": {
      "href": "https://.../sales-items?page=2"
    },
    "next": {
```

```
      "href": "https://.../sales-items?page=4"
    },
  },
  "count": 25,
  "total": 1500,
  "_embedded": {
    "salesItems": [
      {
        "_links": {
          "self": {
            "href": "https://.../sales-items/123"
          }
        },
        "id": 123,
        "name": "Sales item 123"
      },
      {
        "_links": {
          "self": {
            "href": "https://.../sales-items/124"
          }
        },
        "id": 124,
        "name": "Sales item 124"
      },
      .
      .
      .
    ]
  }
}
```

Versioning

You can introduce a new version of an API using a versioning URL path segment. Below are example endpoints for API version 2:

```
GET /sales-item-service/v2/sales-items
POST /sales-item-service/v2/sales-items
...
```

Documentation

If you need to document or provide interactive online documentation for a REST API, there are two ways:

1. First, create a specification for the API and then generate code from the specification
2. First, implement the API and then generate the API specification from the code

Tools like *Swagger* and *Postman* can generate both static and interactive documentation for your API based on the API specification. You should specify APIs using the *OpenAPI* specification (https://swagger.io/specification/).

When using the first alternative, you can specify your API using the OpenAPI specification language. You can use tools like SwaggerHub or Postman to write the API spec. Swagger offers code-generation tools for multiple languages. Code generators generate code based on the OpenAPI spec.

When using the second alternative, you can use a web framework-specific way to build the API spec from the API implementation. For example, if you are using Spring Boot, you can use the *springdoc-openapi-ui* library, and with Nest.js, you can use the *@nestjs/swagger* library.

Implementation Example

Let's implement the API endpoints for the *sales-item-service* using Typescript and Nest.js and its `Controller` decorator:

SalesItemController.ts

```
import {
  Controller,
  Get,
  Query,
  Post,
  Body,
  Put,
  Param,
  Delete
} from '@nestjs/common';
// ...

@Controller('sales-items')
export class SalesItemController {

  constructor(private readonly salesItemService: SalesItemService) {}

  @Post()
  create(@Body() salesItemArg: SalesItemArg): Promise<SalesItem> {
    return this.salesItemService
      .createSalesItem(salesItemArg);
  }

 @Get()
  getSalesItems(
    @Query('userAccountId') userAccountId: string,
  ): Promise<SalesItem[]> {
    if (userAccountId) {
      return this.salesItemService.getSalesItemsByUserAccountId(
        parseInt(userAccountId, 10),
      );
    } else {
      return this.salesItemService.getSalesItems();
    }
  }

  @Get('/:id')
  getSalesItemById()(@Param('id') id: string): Promise<SalesItem> {
    return this.salesItemService
      .getSalesItemById(parseInt(id, 10));
  }
```

```
  @Put('/:id')
  @HttpCode(204)
  updateSalesItem(
    @Param('id') id: string,
    @Body() salesItemArg: SalesItemArg
  ):Promise<void> {
    return this.salesItemService
      .updateSalesItem(parseInt(id, 10), salesItemArg);
  }

  @Delete('/:id')
  @HttpCode(204)
  deleteSalesItemById(@Param('id') id: string): Promise<void> {
    return this.salesItemService
      .deleteSalesItemById(parseInt(id, 10));
  }

  @Delete()
  @HttpCode(204)
  deleteSalesItems(): Promise<void> {
    return this.salesItemService.deleteSalesItems();
  }
}
```

Next, we create the `SalesItemService` class, which will be injected into the controller by the Nest.js framework. To make it an injectable dependency, we decorate the `SaleItemService` class with the `@Injectable` decorator. We list the `SaleItemService` class in the application module's providers to allow an instance of the `SalesItemService` class to be injected where needed, e.g., into an instance of the `SalesItemController` class.

SalesItemService.ts

```
import { Injectable } from '@nestjs/common';
// ...

@Injectable()
export class SalesItemService {
  // Implement service methods...
}
```

AppModule.ts

```
import { Module } from '@nestjs/common';
// ...

@Module({
  controllers: [SalesItemController],
  providers: [SalesItemService],
})
export class AppModule {}
```

Regarding errors, Nest.js instructs to throw an instance of its `HttpException` class or an instance of a `HttpException` subclass. This is not the correct way to handle errors. Errors should not be tied to a specific web framework. You should throw business errors from your business logic and use

your web framework's error mapping/handling/filtering functionality to map business errors to HTTP responses. The best way to achieve this is to create a base error class that can store an HTTP status code, possibly an error code, and, of course, the error message itself.

ApiError.ts

```
export class ApiError extends Error {
  getStatusCode(): number {
    return 500;
  }

  getErrorCode(): string {
   return "UNSPECIFIED";
  }
}
```

EntityNotFoundError.ts

```
export class EntityNotFoundError extends ApiError {
  constructor(entityType: string, id: number) {
    super(`${entityType} entity not found with id ${id}`);
  }

  override getStatusCode(): number {
    return 404;
  }
}
```

GraphQL API Design

> *Divide API endpoints into queries and mutations. Compared to REST, REST GET requests are GraphQL queries, and REST POST/PUT/PATCH/DELETE requests are GraphQL mutations. With GraphQL, you can name your queries and mutations with descriptive names.*

Let's create a GraphQL schema that defines needed types and API endpoints for the *sales-item-service*:

schema.graphql

```
type Image {
  id: ID!
  rank: Int!
  url: String!
}

type SalesItem {
  id: ID!
  createdAtTimestampInMillis: String!
  name: String!
  price: Float!
  images(
    sortByField: String = "rank",
    sortDirection: SortDirection = ASC,
    offset: Int = 0,
```

```
    limit: Int = 5
  ): [Image!]!
}

input InputImage {
  id: ID!
  rank: Int!
  url: String!
}

input NewSalesItem {
  name: String!
  price: Float!
  images: [InputImage!]!
}

input UpdatedSalesItem {
  name: String
  price: Float
  images: [InputImage!]
}

enum SortDirection {
  ASC,
  DESC
}

type IdResponse {
  id: ID!
}

type Query {
  salesItems(
    sortByField: String = "createdAtTimestamp",
    sortDirection: SortDirection = DESC,
    offset: Int = 0,
    limit: Int = 50
  ): [SalesItem!]!

  salesItem(id: ID!): SalesItem!

  salesItemsByFilters(
    nameContains: String,
    priceGreaterThan: Float
  ): [SalesItem!]!
}

type Mutation {
  createSalesItem(salesItem: NewSalesItem!): SalesItem!

  updateSalesItem(
    id: ID!,
    updatedSalesItem: UpdatedSalesItem
  ): IdResponse!

  deleteSalesItem(id: ID!): IdResponse!
}
```

In the above GraphQL schema, we define several types used in API requests and responses. A GraphQL `type` specifies what properties the type has and the types of those properties. A type specified with the `input` keyword is an input-only type (input DTO type). GraphQL defines the

following primitive (scalar) types: `Int` (32-bit), `Float`, `String`, `Boolean`, and `ID`. You can define an array type with the notation: `[<Type>]`. By default, types are nullable. If you want a non-nullable type, you must add an exclamation mark (!) after the type name. You can define an enumerated type with the `enum` keyword. The `Query` and `Mutation` types are special GraphQL types used to define queries and mutations. The above example defines three queries and four mutations that clients can execute. You can add parameters for a type property. We have added parameters for all the queries (queries are properties of the `Query` type), mutations (mutations are properties of the `Mutation` type), and the `images` property of the `SalesItem` type.

In the above example, I have named all the queries with names that describe the values they return, i.e., there are no verbs in the query names. It is possible to name queries starting with a verb (like the mutations). For example, you could add *get* to the names of the above-defined queries if you wanted.

The Apollo server below implements some GraphQL type resolvers returning static responses. After starting the server with the `node server.js` command, you can browse to *http://localhost:4000* and try to execute some of the implemented queries or mutations.

server.js

```
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';

const typeDefs = readFileSync('./schema.graphql',
                              { encoding: 'utf8' });

const resolvers = {
  Query: {
    salesItems: (_, { sortByField,
                      sortDirection,
                      offset,
                      limit }) =>
    [{
        id: 1,
        createdAtTimestampInMillis: '12345678999877',
        name: 'sales item',
        price: 10.95
    }],
    salesItem: (_, { id }) => ({
      id,
      createdAtTimestampInMillis: '12345678999877',
      name: 'sales item',
      price: 10.95
    })
  },
  Mutation: {
    createSalesItem: (_, { newSalesItem }) => {
      return {
        id: 100,
        createdAtTimestampInMillis: Date.now().toString(),
        ...newSalesItem
      };
    },
    deleteSalesItem: (_, { id }) => {
      return {
```

```
        id
      };
    }
  },
  SalesItem: {
    images: (parent) => {
      return [{
        id: 1,
        rank: 1,
        url: 'url'
      }];
    }
  }
};

const server = new ApolloServer({
  typeDefs,
  resolvers
});

startStandaloneServer(server, {
  listen: { port: 4000 }
});
```

GraphQL error handling differs from REST API error handling. A GraphQL API does not provide different HTTP response status codes. When an error happens while processing a GraphQL API request, the response body object will include an `errors` array. In your GraphQL type resolvers, you should throw a `GraphQLError` object when a query or mutation fails. By supplying an `extensions` object, you can supply custom error properties when throwing a `GraphQLError` object. In the below example, we supply an `errorCode` property in the `extensions` object.

```
import { GraphQLError } from 'graphql';

throw new GraphQLError(message, {
  extensions: { errorCode: 'YOUR_ERROR_CODE' }
});
```

If you want, you can supply an HTTP status code in the `extensions` object, too:

```
import { GraphQLError } from 'graphql';

throw new GraphQLError(message, {
  extensions: {
    statusCode: 500,
    status: 'Internal Server Error',
    errorCode: 'YOUR_ERROR_CODE'
  }
});
```

Remember to follow the *clean microservice design principle* when implementing production code. Use controllers, services, repositories, DTOs, and entities. Below is an example of a GraphQL controller:

GraphQlSalesItemController.js

```
export default class GraphQlSalesItemController {
  salesItemService;
  salesItemImageService;

  // ...

  getSalesItems(...) {
    return this.salesItemService.getSalesItems(...);
  }

  getSalesItem({ id }) {
    return this.salesItemService.getSalesItem(id);
  }

  getSalesItemImages({ id }) {
    return this.salesItemImageService.getSalesItemImages(id);
  }

  createSalesItem({ newSalesItem }) {
    return this.salesItemService.createSalesItem(newSalesItem);
  }

  deleteSalesItem({ id }) {
    this.salesItemService.deleteSalesItem(id);
    return { id };
  }

  getResolvers() {
    return {
      Query: {
        salesItems: this.getSalesItems,
        salesItem: this.getSalesItem
      },
      Mutation: {
        createSalesItem: this.createSalesItem,
        deleteSalesItem: this.deleteSalesItem
      },
      SalesItem: {
        images: this.getSalesItemImages
      }
    }
  }
}
```

You can use the above controller (and other possible controllers) to bootstrap your GraphQL server:

server.js

```
import { merge } from 'lodash';
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';
import GraphQlSalesItemController
  from './GraphQlSalesItemController.js';
// ...

const typeDefs = readFileSync('./schema.graphql',
                              { encoding: 'utf8' });

const resolvers = merge(
  {},
```

```
  [
    new GraphQlSalesItemController(),
    new GraphQlUserController(),
    new GraphQlOrderController()
  ].map(controller => controller.getResolvers())
);

const server = new ApolloServer({
  typeDefs,
  resolvers
});

startStandaloneServer(server, {
  listen: { port: 4000 }
});
```

In a GraphQL schema, you can add parameters for a primitive (scalar) property, also. That is useful for implementing conversions.

For example, we could define the `SalesItem` type with a parameterized `price` property:

```
enum Currency {
  USD,
  GBP,
  EUR,
  JPY
}

type SalesItem {
  id: ID!
  createdAtTimestampInMillis: String!
  name: String!
  price(currency: Currency = USD): Float!
  images(
    sortByField: String = "rank",
    sortDirection: SortDirection = ASC,
    offset: Int = 0,
    limit: Int = 5
  ): [Image!]!
}
```

Now clients can supply a currency parameter for the `price` property in their queries to get the price in different currencies. The default currency is *USD* if no currency parameter is supplied.

Below are two example queries that a client could perform against the earlier defined GraphQL schema:

```
{
  # gets the name, price in euros and the first 5 images
  # for the sales item with id "1"
  salesItem(id: "1") {
    name
    price(currency: EUR)
    images
  }

  # gets the next 5 images for the sales item 1
```

```
  salesItem(id: "1") {
    images(offset: 5)
  }
}
```

In real life, consider limiting the fetching of resources only to the previous or the next page (or the next page only if you are implementing infinite scrolling on the client side). Then, clients cannot fetch random pages. This prevents attacks where a malicious user tries to fetch pages with huge page numbers (like 10,000, for example) which can cause extra load for the server or, at the extreme, a denial of service.

Below is an example where clients can only query the first, next, or previous page. When a client requests the first page, the page cursor can be empty, but when the client requests the previous or the next page, it must give the current page cursor as a query parameter.

```
type PageOfSalesItems {
  # Contains the page number encrypted and
  # encoded as a Base64 value.
  pageCursor: String!

  salesItems: [SalesItem!]!
}

enum Page {
  FIRST,
  NEXT,
  PREVIOUS
}

type Query {
  pageOfSalesItems(
    page: Page = FIRST,
    pageCursor: String = ""
  ): PageOfSalesItems!
}
```

In the above GraphQL examples, we did not implement user input validation. Validation is required in production code. You can manually add validation logic to each resolver function, and if input parameters are not valid, you can throw a validation error from the resolver.

One major shortcoming in the GraphQL schema language is that you cannot define validation. Even though validation is always needed for `input` types. Consider implementing a GraphQL API with TypeScript instead of JavaScript. Then you can use the *type-graphql* NPM library that allows you to write a GraphQL schema using TypeScript classes with decorator-based validation. Below is the `NewSalesItem` input type from the earlier GraphQL schema represented as a TypeScript class with validation decorators from the *class-validator* library. When using the *type-graphql* library, you don't have to write validation logic to each resolver function separately. Validation is handled by the *class-validator* library transparently, and your code will be cleaner. The *type-graphql* library works with most GraphQL server implementations, like *express-graphql* or *apollo-server*.

NewSalesItem.ts

```
import { Field, InputType } from 'type-graphql';
import { Max, MaxLength, Min } from "class-validator";
import InputImage from './InputImage';

@InputType()
export default class NewSalesItem {
  @Field()
  @MaxLength(256)
  title: name;

  @Field()
  @Min(0)
  @Max(100000000)
  price: number;

  @Field(type => [InputImage])
  images: InputImage[];
}
```

Instead of *type-graphql*, you can use the Nest.js web framework. It also allows you to define a GraphQL schema using TypeScript classes and validation decorators. The above and below examples are identical, except that some decorators are imported from a different library.

NewSalesItem.ts

```
import { Field, Int, InputType } from '@nestjs/graphql';
import { Max, MaxLength, Min } from "class-validator";
import InputImage from './InputImage';

@InputType()
export class NewSalesItem {
  @Field()
  @MaxLength(256)
  title: name;

  @Field()
  @Min(0)
  @Max(100000000)
  price: number;

  @Field(type => [InputImage])
  images: InputImage[];
}
```

Both with *type-graphql* and Nest.js, you don't specify a controller for GraphQL queries and mutations. You specify resolver classes that are used to compose the final root resolver for the GraphQL API. Below is an example Nest.js resolver:

SalesItemResolver.ts

```
import {
  Args,
  Int,
  Parent,
  Query,
  ResolveField,
  Resolver,
```

```
} from '@nestjs/graphql';
// ...

@Resolver(of => SalesItem)
export class SalesItemResolver {
  constructor(
    private salesItemService: SalesItemService,
    private salesItemImageService: SalesItemImageService
  ) {}

  @Query(returns => SalesItem)
  async salesitem(@Args('id', { type: () => Int }) id: number) {
    return this.salesItemService.getSalesItem(id);
  }

  @ResolveField()
  async images(@Parent() salesItem: SalesItem) {
    return this.salesItemImageService
      .getSalesItemImages(salesItem.id);
  }
}
```

Subscription-Based API Design

Design a subscription-based API when you want clients to be able to subscribe to small, incremental changes to large objects or when clients want to receive low-latency real-time updates.

Server-Sent Events (SSE)

Server-Sent Events (SSE) is a uni-directional push technology enabling a client to receive updates from a server via an HTTP connection.

Let's showcase the SSE capabilities with a real-life example. The below example defines a *subscribe-to-loan-application-summaries* API endpoint for clients to subscribe to loan application summaries. A client will show loan application summaries in a list view in its UI. Whenever there is a new summary for a loan application available, the server will send a loan application summary event to clients that will update their UIs by adding a new loan application summary.

server.js

```
import express from 'express';
import bodyParser from 'body-parser';
import loanApplicationSummariesSubscriptionHandler
  from './loanApplicationSummariesSubscriptionHandler.js';

const app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));

app.get('/subscribe-to-loan-application-summaries',
        loanApplicationSummariesSubscriptionHandler);

app.listen(3001);
```

subscribers.js

```
import { v4 as uuidv4 } from 'uuid';

export let subscribers = [];

export function addSubscriber(response) {
  const id = uuidv4();

  const subscriber = {
    id,
    response
  };

  subscribers.push(subscriber);
  return id;
}

export function removeSubscriber(id) {
  subscribers = subscribers.filter((subscriber) =>
    subscriber.id !== id);
}
```

loanApplicationSummariesSubscriptionHandler.js

```
import {
  addSubscriber,
  removeSubscriber
} from './subscribers.js';

export default function
loanApplicationSummariesSubscriptionHandler(request, response) {
  // Response headers needed for SSE:
  // - Server sent events are identified with
  //   content type 'text/event-stream'
  // - The connection must be kept alive so that server
  //   can send continuously data to client
  // - Server sent events should not be cached
  const headers = {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive',
    'Cache-Control': 'no-cache'
    // For dev environment you can add CORS header:
    'Access-Control-Allow-Origin': '*'
  };

  response.writeHead(200, headers);

  // Server sent event must be a string beginning with 'data: '
  // and ending with two newline characters
  // First event is empty
  const data = 'data: \n\n';
  response.write(data);
  const subscriberId = addSubscriber(response);
  request.on('close', () => removeSubscriber(subscriberId));
}
```

The below `publishLoanApplicationSummary` function is called whenever the server receives a new loan application summary. The server can receive loan application summaries as messages consumed from a message broker's topic. (This message consumption part is not implemented

here, but there is another example later in this chapter demonstrating how messages can be consumed from a Kafka topic.)

publishLoanApplicationSummary.js

```
import { subscribers } from './subscribers.js';

export default function publishLoanApplicationSummary(
  loanApplicationSummary
) {
   // Send an event to each subscriber
   // Loan application summary data is converted to JSON
   // before sending the event
   // Server sent event must be a string beginning with 'data: '
   // and ending with two newline characters
   const data = JSON.stringify(loanApplicationSummary);
   subscribers.forEach(({ response }) =>
     response.write(`data: ${data}\n\n`));
}
```

Next, we can implement the web client and define the following React functional component:

LoanApplicationSummaries.jsx

```
import React, { useEffect, useState } from 'react';

export default function LoanApplicationSummaries() {
  const [ loanApplicationSummaries,
          setLoanApplicationSummaries ] = useState([]);

  // Define an effect to be executed on component mount
  useEffect(() => {
    // Create new event source
    // Hardcoded dev environment URL is used here for demonstration
    // purposes
    const eventSource =
      new EventSource('http://localhost:3001/subscribe-to-loan-application-summaries');

    // Listen to server sent events and add a new
    // loan application summary to the head of
    // loanApplicationSummaries array
    eventSource.addEventListener('message', (messageEvent) => {
      try {
        const loanApplicationSummary =
          JSON.parse(messageEvent.data);

        if (loanApplicationSummary) {
          setLoanApplicationSummaries([loanApplicationSummary,
                                       ...loanApplicationSummaries]);
        }
      } catch {
        // Handle error
      }
    });


    eventSource.addEventListener('error', (errorEvent) => {
      // Handle error
    });

    // Close the event source on component unmount
```

```
    return function cleanup() { eventSource.close(); }
  }, [loanApplicationSummaries]);

  // Render loan application summary list items
  const loanApplicationSummaryListItems =
    loanApplicationSummaries.map(({ ... }) =>
       (<li key={key here...}>render here...</li>));

  return (
    <ul>{loanApplicationSummaryListItems}</ul>
  );
}
```

GraphQL Subscriptions

Let's have an example of a GraphQL subscription. The below GraphQL schema defines one subscription for a post's comments. It is not relevant what a post is. It can be a blog post or social media post, for example. We want a client to be able to subscribe to a post's comments.

```
type PostComment {
  id: ID!,
  text: String!
}

type Subscription {
  postComment(postId: ID!): PostComment
}
```

On the client side, we can define a subscription named `postCommentText` that subscribes to a post's comments and returns the text property of comments:

```
import { gql } from '@apollo/client';

const POST_COMMENT_SUBSCRIPTION = gql`
  subscription postCommentText($postId: ID!) {
    postComment(postID: $postId) {
      text
    }
  }
`;
```

If a client executes the above query for a particular post (defined with the `postId` parameter), the following kind of response can be expected:

```
{
  "data": {
    "postComment": {
      "text": "Nice post!"
    }
  }
}
```

To be able to use GraphQL subscriptions, you must implement support for them both on the server and client side. For the server side, you can find instructions for the *Apollo server*: https://

www.apollographql.com/docs/apollo-server/data/subscriptions/#enabling-subscriptions. And for the client side, you can find instructions for the *Apollo client* here: https://www.apollographql.com/docs/react/data/subscriptions/#setting-up-the-transport.

After the server and client-side support for subscriptions are implemented, you can use the subscription in your React component:

SubscribedPostCommentsView.jsx

```
import { useState } from 'react';
import { gql, useSubscription } from '@apollo/client';

const POST_COMMENT_SUBSCRIPTION = gql`
  subscription subscribeToPostComment($postId: ID!) {
    postComment(postID: $postId) {
      id
      text
    }
  }
`;

export default function SubscribedPostCommentsView({ postId }) {
  const [ postComments, setPostComments ] = useState([]);

  const { data } = useSubscription(POST_COMMENT_SUBSCRIPTION,
                                   { variables: { postId } });

  if (data?.postComment) {
    setPostComments([...postComments, data.postComment]);
  }

  const postCommentListItems =
    postComments.map(( { id, text }) =>
      (<li key={id}>{text}</li>));

  return <ul>{postCommentListItems}</ul>;
}
```

WebSocket Example

Below is a chat messaging application consisting of a WebSocket server implemented with Node.js and the *ws* NPM library, and a WebSocket client implemented with React. There can be multiple instances of the server running. These instances are stateless except for storing WebSocket connections for locally connected clients. First, we list the source code files for the server side.

A new Kafka client is created using the *kafkajs* NPM library:

kafkaClient.js

```
import { Kafka } from 'kafkajs';

const kafkaClient = new Kafka({
  clientId: 'app-y',
  brokers: [process.env.KAFKA_BROKER],
});

export default kafkaClient;
```

A new Redis client is created using the *ioredis* NPM library:

redisClient.js

```
import Redis from 'ioredis';

const redisClient = new Redis({
  port: parseInt(process.env.REDIS_PORT, 10),
  host: process.env.REDIS_HOST,
  username: process.env.REDIS_USERNAME,
  password: process.env.REDIS_PASSWORD
});

export default redisClient;
```

The below `KafkaMessageBrokerAdminClient` class is used to create topics in Kafka:

KafkaMessageBrokerAdminClient.js

```
export default class KafkaMessageBrokerAdminClient {
  kafkaAdminClient;

  constructor(kafkaClient) {
    this.kafkaAdminClient = kafkaClient.admin();
  }

  async create(topic) {
    try {
      await this.kafkaAdminClient.connect();

      await this.kafkaAdminClient.createTopics({
        topics: [{ topic }]
      });

      await this.kafkaAdminClient.disconnect();
    } catch {
      // Handle error
    }
  }
}
```

Users of the chat messaging application are identified with phone numbers. On the server side, we store the WebSocket connection for each user in the `phoneNbrToWsConnectionMap`:

phoneNbrToWsConnectionMap.js

```
const phoneNbrToWsConnectionMap = new Map();
export default phoneNbrToWsConnectionMap;
```

The below `WebSocketChatMessagingServer` class handles the construction of a WebSocket server. The server accepts connections from clients. When it receives a chat message from a client, it will first parse and validate it. If the received chat message is a special online indication message, the server will register a new user. For an actual chat message, the server will store the message in persistent storage (using a separate *chat-message-service* REST API, not implemented here). The server gets the recipient's server information from a Redis cache and sends the chat message to the recipient's WebSocket connection or produces the chat message to a Kafka topic where

another server instance can consume the chat message and send it to the recipient's WebSocket connection. The Redis cache stores a hash map where the users' phone numbers are mapped to the server instance they are currently connected. A UUID identifies a server instance.

WebSocketChatMessagingServer.js

```
import { WebSocketServer } from 'ws';
import kafkaClient from './kafkaClient.js';
import redisClient from './redisClient.js';

import phoneNbrToWsConnectionMap
  from './phoneNbrToWsConnectionMap.js';

import KafkaMessageBrokerProducer
  from './KafkaMessageBrokerProducer.js';

export default class WebSocketChatMessagingServer {
  wsServer;
  serverUuid;
  messageBrokerProducer;
  wsConnectionToPhoneNbrMap = new Map();

  constructor(serverInstanceUuid) {
    this.serverUuid = serverUuid;

    this.messageBrokerProducer =
      new KafkaMessageBrokerProducer(kafkaClient);

    this.wsServer = new WebSocketServer({ port: 8080 });

    this.wsServer.on('connection', wsConnection => {
      wsConnection.on('message', async (chatMessageJson) => {
        const chatMessage = this.parse(chatMessageJson);

        if (chatMessage) {
          const messageIsOnlineIndicator =
            chatMessage.message === 'online' &&
            !chatMessage.recipientPhoneNbr;

          if (messageIsOnlineIndicator) {
            await this.handleOnlineIndication(
              wsConnection,
              chatMessage.senderPhoneNbr
            );
          } else {
            this.store(chatMessage);

            const recipientServerUuid =
              await this.getServerUuid(
                chatMessage.recipientPhoneNbr
              );

            this.send(chatMessage, recipientServerUuid);
          }
        }
      });

      wsConnection.on('close', () => {
        this.close(wsConnection);
      });
    });
  }
```

```
closeServer() {
  this.wsServer.close();
  this.wsServer.clients.forEach(client => client.close());
  this.messageBrokerProducer.close();
}

parse(chatMessageJson) {
  return (() => {
    try {
      const chatMessage = JSON.parse(chatMessageJson);
      // Validate chatMessage properties here
      return chatMessage;
    } catch {
      return '';
    }
  })();
}

async handleOnlineIndication(
  wsConnection,
  senderPhoneNbr
) {
  phoneNbrToWsConnectionMap.set(
    senderPhoneNbr,
    wsConnection
  );

  this.wsConnectionToPhoneNbrMap.set(
    wsConnection,
    senderPhoneNbr
  );

  try {
    await redisClient.hset(
      'phoneNbrToServerUuidMap',
      senderPhoneNbr,
      this.serverUuid
    );
  } catch {
    // Handle error
  }
}

store(chatMessage) {
  fetch('http://.../chat-message-service/chat-messages',
    { method: 'post',
      body: JSON.stringify(chatMessage)
    })
    .catch(() => {
      // Handle error
    });
}

async getServerUuid(phoneNbr) {
  try {
    return
      await redisClient.hget(
        'phoneNbrToServerUuidMap',
         phoneNbr
      );
  } catch {
    return undefined;
```

```
    }
  }

  send(chatMessage, serverUuid) {
    if (serverUuid === this.serverUuid) {
      // Recipient has active connection on
      // the same server instance as sender
      const recipientWsConnection =
        phoneNbrToWsConnectionMap
          .get(chatMessage.recipientPhoneNbr);

      recipientWsConnection?
        .send(JSON.stringify(chatMessage));

    } else if (serverUuid) {
      // Recipient has active connection on different
      // server instance compared to sender
      const serverTopic = serverUuid;

      this.messageBrokerProducer
        .produce(chatMessage, serverTopic);
    }
  }

  async close(wsConnection) {
    const phoneNbr =
      this.wsConnectionToPhoneNbrMap
        .get(wsConnection);

    phoneNbrToWsConnectionMap.delete(phoneNbr);
    this.wsConnectionToPhoneNbrMap.delete(wsConnection);

    try {
      await redisClient.hdel('phoneNbrToServerUuidMap',
                             phoneNbr);
    } catch {
      // Handle error
    }
  }
}
```

KafkaMessageBrokerProducer.js

```
export default class KafkaMessageBrokerProducer {
  kafkaProducer;

  constructor(kafkaClient) {
    this.kafkaProducer = kafkaClient.producer();
  }

  async produce(chatMessage, topic) {
    try {
      await this.kafkaProducer.connect();
      await this.kafkaProducer.send({
        topic,
        messages: [{ value: JSON.stringify(chatMessage) }],
      });
    } catch {
      // Handle error
    }
  }

  async close() {
```

```
    try {
      this.isTerminating = true;
      await this.kafkaProducer.disconnect();
    } catch {
    }
  }
}
```

The `KafkaMessageBrokerConsumer` class defines a Kafka consumer that consumes chat messages from a particular Kafka topic and sends them to the recipient's WebSocket connection:

KafkaMessageBrokerConsumer.js

```
import phoneNbrToWsConnectionMap
  from './phoneNbrToWsConnectionMap.js';

export default class KafkaMessageBrokerConsumer {
  kafkaConsumer;

  constructor(kafkaClient) {
    this.kafkaConsumer =
      kafkaClient.consumer({ groupId: 'app-y' });
  }

  async consumeChatMessages(topic) {
    await this.kafkaConsumer.connect();
    await this.kafkaConsumer.subscribe({
      topic,
      fromBeginning: true
    });

    this.kafkaConsumer.run({
      eachMessage: async ({ message }) => {
        try {
          const chatMessage =
            JSON.parse(message.value.toString());

          const recipientWsConnection =
            phoneNbrToWsConnectionMap
              .get(chatMessage.recipientPhoneNbr);

          recipientWsConnection?
            .send(JSON.stringify(chatMessage));
        } catch {
          // Handle error
        }
      },
    });
  }

  async close() {
    try {
      await this.kafkaConsumer.disconnect();
    } catch
    {}
  }
}
```

Finally, we put it all together in the index.js file:

index.js

```
import { v4 as uuidv4 } from 'uuid';
import kafkaClient from './kafkaClient.js';

import KafkaMessageBrokerAdminClient
  from './KafkaMessageBrokerAdminClient.js';

import KafkaMessageBrokerConsumer
  from './KafkaMessageBrokerConsumer.js';

import WebSocketChatMessagingServer
  from './WebSocketChatMessagingServer.js';

// Generate a unique id for this particular server instance
const serverUuid = uuidv4();
const serverTopic = serverUuid;

// Create a Kafka topic for this particular microservice instance
await new KafkaMessageBrokerAdminClient(kafkaClient)
  .create(serverTopic);

// Create the chat messaging service for
// handling WebSocket connections
const chatMessagingServer =
  new WebSocketChatMessagingServer(serverUuid);

// Create and start a Kafka consumer to consume and send
// chat messages for recipients that are connected to
// this microservice instance
const messageBrokerConsumer =
  new KafkaMessageBrokerConsumer(kafkaClient);

messageBrokerConsumer.consumeChatMessages(serverTopic);

// Close the Web Socket server and Kafka consumer before exiting
function prepareExit() {
  chatMessagingServer.closeServer();
  messageBrokerConsumer.close();
}

// Handle signals and prepare for the process exit
process.once('SIGINT', prepareExit);
process.once('SIGQUIT', prepareExit);
process.once('SIGTERM', prepareExit);
```

For the web client, we have the below code. An instance of the `ChatMessagingService` class connects to a chat messaging server via WebSocket. It listens to messages received from the server and dispatches an action upon receiving a chat message. The class also offers a method for sending a chat message to the server.

ChatMessagingService.js

```
import store from "./store";

class ChatMessagingService {
  wsConnection;
  connectionIsOpen = false;
  lastChatMessage;

  constructor(dispatch, userPhoneNbr) {
```

```
    this.wsConnection = new WebSocket('ws://localhost:8080');

    this.wsConnection.addEventListener('open', () => {
      this.connectionIsOpen = true;

      const onlineMessage = {
        message: 'online',
        senderPhoneNbr: userPhoneNbr
      };

      this.wsConnection.send(JSON.stringify(onlineMessage));
    });

    this.wsConnection.addEventListener('error', () => {
      this.lastChatMessage = null;
    });

    this.wsConnection.addEventListener(
      'message',
      ({ data: chatMessageJson }) => {
        const chatMessage = JSON.parse(chatMessageJson);

        store.dispatch({
          type: 'receivedChatMessageAction',
          chatMessage
        });
      });

    this.wsConnection.addEventListener('close', () => {
      this.connectionIsOpen = false;
    });
  }

  send(chatMessage) {
    this.lastChatMessage = chatMessage;

    if (this.connectionIsOpen) {
        this.wsConnection.send(JSON.stringify(chatMessage));
    } else  {
      // Send message to REST API
    }
  }

  close() {
    this.connectionIsOpen = false;
    this.wsConnection.close();
  }
}

export let chatMessagingService;

export default function createChatMessagingService(
  userPhoneNbr
) {
  chatMessagingService =
    new ChatMessagingService(store.dispatch, userPhoneNbr);

  return chatMessagingService;
}
```

index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux'
import ChatApp from './ChatApp';
import store from './store'
import './index.css';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(
  <Provider store={store}>
    <ChatApp/>
  </Provider>
);
```

The chat application parses the user's and contact's phone numbers from the URL and then renders a chat view between the user and the contact:

ChatApp.jsx

```
import React, { useEffect } from 'react';
import queryString from "query-string";
import ContactChatView from "./ContactChatView";
import createChatMessagingService from "./ChatMessagingService";

const { userPhoneNbr, contactPhoneNbr } =
  queryString.parse(window.location.search);

export default function ChatApp() {
  useEffect(() => {
    const chatMessagingService =
      createChatMessagingService(userPhoneNbr);

    return function cleanup() {
      chatMessagingService.close();
    }
  }, []);

  return (
    <div>
      User: {userPhoneNbr}
      <ContactChatView
        userPhoneNbr={userPhoneNbr}
        contactPhoneNbr={contactPhoneNbr}
      />
    </div>
  );
}
```

The `ContactChatView` component renders chat messages between a user and a contact:

ContactChatView.jsx

```
import React, { useEffect, useRef } from 'react';
import { connect } from "react-redux";
import store from './store';

function ContactChatView({
  userPhoneNbr,
  contactPhoneNbr,
  chatMessages,
  fetchLatestChatMessages
}) {
  const inputElement = useRef(null);

  useEffect(() => {
    fetchLatestChatMessages(userPhoneNbr, contactPhoneNbr);
  }, [contactPhoneNbr,
      fetchLatestChatMessages,
      userPhoneNbr]
  );

  function sendChatMessage() {
    if (inputElement?.current.value) {
      store.dispatch({
        type: 'sendChatMessageAction',
        chatMessage: {
          senderPhoneNbr: userPhoneNbr,
          recipientPhoneNbr: contactPhoneNbr,
          message: inputElement.current.value
        }
      });
    }
  }

  const chatMessageElements = chatMessages
    .map(({ message, senderPhoneNbr }, index) => {
      const messageIsReceived =
        senderPhoneNbr === contactPhoneNbr;

      return (
        <li
          key={index}
          className={messageIsReceived ? 'received' : 'sent'}>
            {message}
        </li>
      );
    });

  return (
    <div className="contactChatView">
      Contact: {contactPhoneNbr}
      <ul>{chatMessageElements}</ul>
      <input ref={inputElement}/>
      <button onClick={sendChatMessage}>Send</button>
    </div>
  );
}

function mapStateToProps(state) {
  return {
    chatMessages: state
  };
}
```

```
export default connect(mapStateToProps)(ContactChatView);
```

store.js

```
import { createStore } from 'redux';
import { chatMessagingService } from "./ChatMessagingService";

function chatMessagesReducer(state = [], { type, chatMessage }) {
  switch (type) {
    case 'receivedChatMessageAction':
      return state.concat([chatMessage]);
    case 'sendChatMessageAction':
      chatMessagingService.send(chatMessage);
      return state.concat([chatMessage]);
    default:
      return state;
  }
}

const store = createStore(chatMessagesReducer)
export default store;
```

index.css

```
.contactChatView {
  width: 420px;
}

.contactChatView ul {
  padding-inline-start: 0;
  list-style-type: none;
}

.contactChatView li {
  margin-top: 15px;
  width: fit-content;
  max-width: 180px;
  padding: 10px;
  border: 1px solid #888;
  border-radius: 20px;
}

.contactChatView li.received {
  margin-right: auto;
}

.contactChatView li.sent {
  margin-left: auto;
}
```

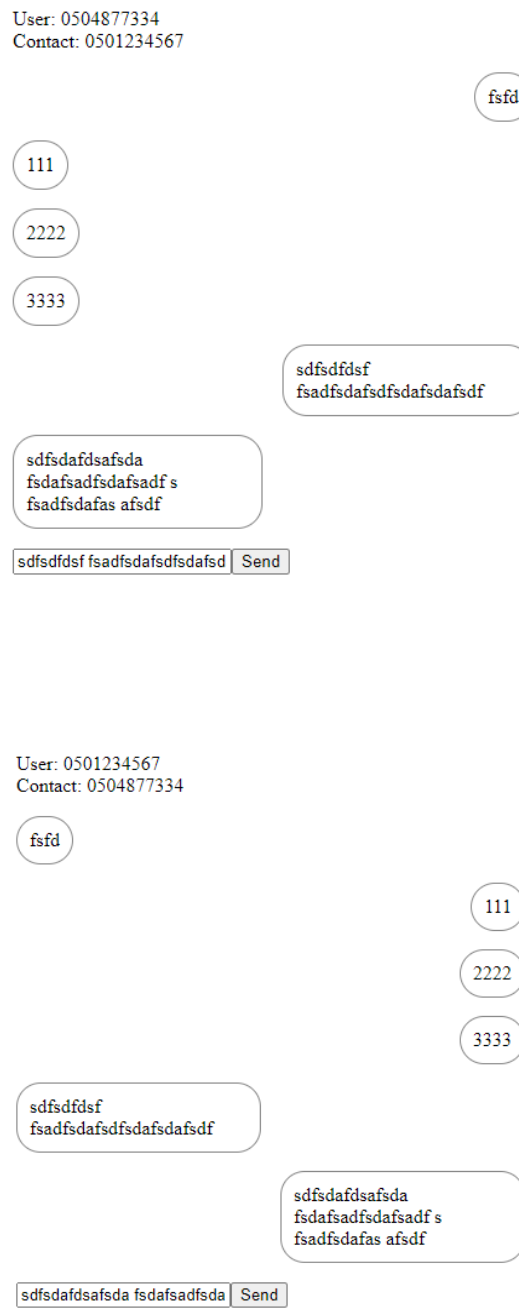

Fig 7.1 Chat Messaging Application Views for Two Users

Inter-Microservice API Design Principles

Inter-microservice APIs can be divided into two categories based on the type of communication: synchronous and asynchronous. Synchronous communication should be used when an immediate response to an issued request is expected. Asynchronous communication should be used when no response to a request is expected, or the response is not immediately required.

Synchronous API Design Principle

> *Use HTTP-based RPC, REST, or GraphQL APIs with JSON data encoding, preferably with HTTP/2 or HTTP/3 transport when requests and responses are not very large and do not contain much binary data. If you have large requests or responses or a lot of binary data, you are better off encoding the data in Avro binary format (Content-Type: avro/binary) instead of JSON or using a gRPC-based API. gRPC always encodes data in a binary format (Protocol Buffers).*

gRPC-Based API Design Example

Let's have an example of a gRPC-based API. First, we must define the needed Protocol Buffers types. They are defined in a file named with the extension *.proto*. The syntax of *.proto* files is pretty simple. We define the service by listing the remote procedures. A remote procedure is defined with the following syntax: `rpc <procedure-name> (<argument-type>) returns (<return-type>) {}`. A type is defined with the below syntax:

```
message <type-name> {
  <field-type> <field-name> [= <field-index>];
  ...
}
```

salesItemService.proto

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.silensoft.salesitemservice";
option java_outer_classname = "SalesItemServiceProto";
option objc_class_prefix = "SIS";

package salesitemservice;

service SalesItemService {
  rpc createSalesItem (NewSalesItem) returns (SalesItem) {}
  rpc getSalesItems(Nothing) returns (SalesItems) {}
  rpc getSalesItem (Id) returns (SalesItem) {}
}

message Nothing {}
```

```
message NewSalesItem {
  string name = 1;
  float price = 2;
}

message SalesItem {
  uint64 id = 1;
  uint64 createdAtTimestampInMillis = 2;
  string name = 3;
  float price = 4;
}

message Id {
  uint64 id = 1;
}

message SalesItems {
  repeated SalesItem salesItem = 1;
}
```

Below is a partial implementation of the gRPC server. We have only implemented the `createSalesItem` procedure.

SalesItemServiceServer.java

```
package com.silensoft.salesitemservice;

import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.concurrent.TimeUnit;
import lombok.Log;

@Log
public class SalesItemServiceServer {
  private Optional<Server> maybeGrpcServer;

  private void start() throws IOException {
    maybeGrpcServer = Optional.of(ServerBuilder.forPort(50051)
      .addService(new SalesItemServiceImpl())
      .build());

    maybeGrpcServer.get().start();
    logger.info("Server started, listening on port: " + port);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      @Override
      public void run() {
        try {
          stop();
        } catch (final InterruptedException exception) {
          exception.printStackTrace(System.err);
        }

        System.err.println("Server shut down");
      }
    });
  }

  private void stop() throws InterruptedException {
```

```
    maybeGrpcServer
      .ifPresent(grpcServer ->
        grpcServer
          .shutdown()
          .awaitTermination(30, TimeUnit.SECONDS));
  }

  private void waitUntilTerminated() throws InterruptedException {
    maybeGrpcServer.ifPresent(Server::awaitTermination);
  }

  public static void main(
    final String[] args
  ) throws IOException, InterruptedException {
    final var server = new SalesItemServiceServer();
    server.start();
    server.waitUntilTerminated();
  }

  static class SalesItemServiceImpl
                extends SalesItemServiceGrpc.ImplBase {
    @Override
    public void createSalesItem(
      final NewSalesItem salesItem,
      final StreamObserver<SalesItem> responseObserver
    ) {
      // Set createdAtTimestamp value on salesItem to be now
      // Insert sales item to database and return 'id'

      salesItem.id = id;
      responseObserver.onNext(salesItem);
      responseObserver.onCompleted();
    }
  }
}
```

Below is the gRPC client implementation for the above gRPC server:

SalesItemServiceClient.java

```
package com.silensoft.salesitemservice;

import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;
import lombok.Log;

@Log
public class SalesItemServiceClient {
  private final SalesItemServiceGrpc.SalesItemServiceBlockingStub salesItemService;

  public SalesItemServiceClient(final Channel channel) {
    salesItemService =
      SalesItemServiceGrpc.newBlockingStub(channel);
  }

  public SalesItem createSalesItem(final NewSalesItem salesItem) {
    try {
```

```
      return salesItemService.createSalesItem(salesItem);
    } catch (final StatusRuntimeException exception) {
       logger.log(Level.ERROR,
                  "CreateSalesItem failed: {0}",
                  exception.getStatus());
    }
  }

  public static void main(final String[] args) throws Exception {
    final var channel = ManagedChannelBuilder
                          .forTarget("localhost:50051")
                          .usePlaintext()
                          .build();

    try {
      SalesItemServiceClient client =
        new SalesItemServiceClient(channel);

      client.createSalesItem(...);
    } finally {
      channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
    }
  }
}
```

Asynchronous API Design Principle

> *Use asynchronous APIs when requests are request-only (fire-and-forget, i.e., no response is expected) or when the response is not immediately expected.*

Request-Only Asynchronous API Design

In request-only asynchronous APIs, the request sender does not expect a response. Such APIs are typically implemented using a message broker. The request sender will send a JSON format request to a topic in the message broker, where the request recipient consumes the request asynchronously.

Different API endpoints can be specified in a request using a `procedure` property, for example. You can name the `procedure` property as you wish, e.g. `action`, `operation`, `apiEndpoint` etc. Parameters for the procedure can be supplied in a `parameters` property. Below is an example request in JSON:

```
{
  "procedure": "<procedure name>",
  "parameters": {
    "parameterName1": <parameter value>,
    "parameterName2": <parameter value>,
    // ...
  }
}
```

Let's have an example with an email-sending microservice that implements a request-only asynchronous API and handles sending of emails. We start by defining a message broker topic for

the microservice. The topic should be named after the microservice, for example: *email-sending-service*.

In the *email-sending-service*, we define the following request schema for an API endpoint that sends an email:

```
{
  "procedure": "sendEmailMessage",
  "parameters": {
    "fromEmailAddress": "...",
    "toEmailAddresses": ["...", "...", ...],
    "subject": "...",
    "message": "..."
  }
}
```

Below is an example request that some other microservice can produce to the *email-sending-service* topic in the message broker:

```
{
  "procedure": "sendEmailMessage",
  "parameters": {
    "fromEmailAddress": "sender@domain.com",
    "toEmailAddresses": ["receiver@domain.com"],
    "subject": "Status update",
    "message": "Hi, Here is my status update ..."
  }
}
```

Request-Response Asynchronous API Design

A request-response asynchronous API microservice receives requests from other microservices and then produces responses asynchronously. Request-response asynchronous APIs are typically implemented using a message broker. The request sender will send a request to a topic where the request recipient consumes the request asynchronously and then produces a response or responses to a message broker topic or topics. Each participating microservice should have a topic named after the microservice in the message broker.

The request format is the same as defined earlier, but the response has a `response` property instead of the `parameters` property. Thus, responses have the following format:

```
{
   "procedure": "<procedure name>",
   "response": {
     "propertyName1": <property value>,
     "propertyName2": <property value>,
     // ...
   }
}
```

Below is an example where a *loan-application-service* requests a *loan-eligibility-assessment-*

service to assess loan eligibility. The *loan-application-service* sends the following JSON-format request to the message broker's *loan-eligibility-assessment-service* topic:

```
{
  "procedure": "assessLoanEligibility",
  "parameters": {
    "userId": 123456789012,
    "loanApplicationId": 5888482223,
    // Other parameters...
  }
}
```

The *loan-eligibility-assessment-service* responds to the above request by sending the following JSON-format response to the message broker's *loan-application-service* topic:

```
{
  "procedure": "assessLoanEligibility",
  "response": {
     "loanApplicationId": 5888482223,
     "isEligible": true,
     "amountInDollars": 10000,
     "interestRate": 9.75,
     "termInMonths": 120
  }
}
```

Below is an example response when the loan application is rejected:

```
{
  "procedure": "assessLoanEligibility",
  "response": {
    "loanApplicationId": 5888482223,
    "isEligible": false
  }
}
```

Alternatively, request and response messages can be treated as events with some data. When we send events between microservices, we have an *event-driven architecture*. Below are the earlier request and response messages written as events:

```
{
  "event": "assessLoanEligibility",
  "data": {
    "userId": 123456789012,
    "loanApplicationId": 5888482223,
    // ...
  }
}
```

```
{
  "event": "LoanApproved",
  "data": {
     "loanApplicationId": 5888482223,
     "isEligible": true,
     "amountInDollars": 10000,
     "interestRate": 9.75,
```

```
    "termInMonths": 120
  }
}
```

```
{
  "procedure": "LoanRejected",
  "response": {
    "loanApplicationId": 5888482223,
    "isEligible": false
  }
}
```

Databases And Database Principles

This chapter presents principles for selecting and using databases. Principles are presented for the following database types:

- Relational databases
- Document databases
- Key-value databases
- Wide column databases
- Search engines

Relational databases are also called SQL databases because accessing a relational database happens via issuing SQL statements. Databases of the other database types are called NoSQL databases because they either don't support SQL at all or they support only a subset of SQL, possibly with some additions and modifications.

Relational Databases

> *Relational databases are multipurpose databases that suit many needs. Choose a relational database if you are not aware of all the needs you have for a database.*

For example, if you don't know what kind of database queries you need now or will need in the future, you should consider using a relational database that is well-suited for different kinds of queries.

Structure of Relational Database

Data in a relational database is organized in the following hierarchy:

- Logical databases/schemas
    - Tables
        - Columns

A table consists of columns and rows. Data in a database is stored as rows in the tables. Each row has a value for each column in the table. If a row does not have a value for a particular column, then a special `NULL` value is used. You can specify if null values are allowed for a column or not.

A microservice should have a single logical database (or schema). Some relational databases have one logical database (or schema) available by default, and in other databases, you must create a logical database (or schema) by yourself.

Use Object Relational Mapper (ORM) Principle

> *Use an object-relational mapper (ORM) to avoid the need to write SQL and to avoid making your microservice potentially vulnerable to SQL injection attacks. Use an ORM to get the database rows automatically mapped to objects that can be serialized to JSON.*

Many languages have ORM frameworks. Java has *Java Persistence API* (JPA, the most famous implementation of which is *Hibernate*), JavaScript/TypeScript has TypeORM, and Python has the Django framework, for example.

An ORM uses entities as building blocks for the database schema. Each entity class in a microservice is reflected as a table in the database. Use the same name for an entity and the database table, except the table name should be plural. Below is an example of a `SalesItem` entity class:

SalesItem.java

```
@Entity
public class SalesItem {
  private Long id;
  private String name;
  private Integer price;
}
```

Store `SalesItem` entities in a table named `salesitems`. In this book, I use case-insensitive database identifiers and write all identifiers in lowercase. The case sensitivity of a database depends on the database and the operating system it is running on. For example, MySQL is case-sensitive only on Linux systems.

The properties of an entity map to columns of the entity table, meaning that the `salesitems` table has the following columns:

- id
- name
- price

Each entity table must have a primary key defined. The primary key must be unique for each row in the table. In the below example, we are using the `@Id` annotation to define the `id` column as the primary key containing a unique value for each row. The `@GeneratedValue` annotation defines that the database should automatically generate a value for the `id` column using the supplied strategy.

SalesItem.java

```
@Entity
@Table(name = "salesitems")
public class SalesItem {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
  private Integer price;
}
```

ORM can create database tables according to entity specifications in code. Below is an example SQL statement for PostgreSQL that an ORM generates to create a table for storing `SalesItem` entities:

```
CREATE TABLE salesitems (
  id BIGINT GENERATED ALWAYS AS IDENTITY,
  name TEXT,
  price INTEGER,
  PRIMARY KEY (id)
);
```

Columns of a table can be specified as unique and not nullable. By default, a column is nullable and is not unique. Below is an example where we define that the `name` column in the `salesitems` table cannot have null values, and values must be unique. We don't want to store sales items with null names, and we want to store sales items having unique names.

SalesItem.java

```
@Entity
@Table(name = "salesitems")
public class SalesItem {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(unique=true, nullable=false)
  private String name;
```

```
  private Integer price;
}
```

When you have a `SalesItem` entity, you can persist it with an instance of JPA's `EntityManager`:

```
entityManager.persist(salesItem);
```

JPA will generate the needed SQL statement on your behalf and execute it. Below is an example SQL statement generated by the ORM to persist a sales item (Remember that the database autogenerates the `id` column).

```
INSERT INTO salesitems (name, price)
VALUES ('Sample sales item', 10);
```

You can search for the created sales item in the database (assuming here that we have a `getId` getter defined):

```
entityManager.find(SalesItem.class, salesItem.getId());
```

For the above operation, the ORM will generate the following SQL query:

```
SELECT id, name, price FROM salesitems WHERE id = 1;
```

Then you can modify the entity and merge it with the entity manager to update the database:

```
salesItem.setPrice(20);
entityManager.merge(salesItem);
```

Finally, you can delete the sales item with the entity manager:

```
entityManager.remove(salesItem);
```

The ORM will execute the following SQL statement:

```
DELETE FROM salesitems WHERE id = 1;
```

Suppose your microservice executes SQL queries that do not include the primary key column in the query's WHERE clause. In that case, the database engine must perform a full table scan to find the wanted rows. Let's say you want to query sales items, the price of which is less than 10. This can be achieved with the below query:

```
// final var price = ...

final TypedQuery<SalesItem> salesItemsQuery = entityManager
 .createQuery("SELECT s FROM salesitems s WHERE s.price < :price",
              SalesItem.class);

usersSalesItemsQuery.setParameter("price", price);

final List<SalesItem> salesItems = salesItemsQuery.getResultList();
```

The database engine must perform a full table scan to find all the sales items where the `price` column has a value below the `price` variable's value. If the database is large, this can be slow. If you perform the above query often, you should optimize those queries by creating an index. For the above query to be fast, we must create an index for the `price` column using the `@Index` annotation inside the `@Table` annotation:

SalesItem.java

```
@Entity
@Table(
  name = "salesitems",
  indexes = @Index(columnList = "price")
)
public class SalesItem {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(unique=true, nullable=false)
  private String name;

  private Integer price;
}
```

Entity/Table Relationships

Tables in a relational database can have relationships with other tables. There are three types of relationships:

- One-to-one
- One-to-many
- Many-to-many

One-To-One/Many Relationships

In this section, we focus on one-to-one and one-to-many relationships. In a one-to-one relationship, a single row in a table can have a relationship with another row in another table. In a one-to-many relationship, a single row in a table can have a relationship with multiple rows in another table.

Let's have an example with an *order-service* that can store orders in a database. Each order consists of one or more order items. An order item contains information about the bought sales item.

```
@Entity
@Table(name = "orders")
public class Order {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // Other order fields ...

  @OneToMany(mappedBy="order")
  private List<OrderItem> orderItems;
}

@Entity
@Table(name = "orderitems")
public class OrderItem {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToOne
  @JoinColumn(name = "salesitemid")
  private SalesItem salesItem;

  @ManyToOne
  @JoinColumn(name = "orderid", nullable = false)
  private Order order;
}
```

Orders are stored in the `orders` table, and order items are stored in the `orderitems` table, which contains a join column named `orderid`. Using this join column, we can map a particular order item to a specific order. Each order item maps to exactly one sales item. For this reason, the `orderitems` table also contains a join column named `salesitemid`. Using this join column, we can map an order item to a sales item.

Below is the SQL statement generated by the ORM for creating the `orderitems` table. The one-to-one and one-to-many relationships are reflected in the foreign key constraints:

- `fksalesitem`: a `salesitemid` column value in the `orderitems` table references an `id` column value in the `salesitems` table
- `fkorder`: an `orderid` column value in the `orderitems` table references an `id` column value in the `orders` table

```
CREATE TABLE orderitems (
  id BIGINT GENERATED ALWAYS AS IDENTITY,
  salesitemid BIGINT,
  orderid BIGINT,
  CONSTRAINT fksalesitem FOREIGN KEY (salesitemid)
    REFERENCES salesitems(id),
  CONSTRAINT fkorder FOREIGN KEY (orderid)
    REFERENCES orders(id)
```

```
);
```

The following SQL query is executed by the ORM to fetch the order with id 123 and its order items:

```
SELECT o.id, s.name, ...
FROM orders o
LEFT JOIN orderitems oi ON o.id = oi.orderid
LEFT JOIN salesitems s ON s.id = oi.salesitemid
WHERE o.id = 123;
```

Many-To-Many Relationships

In a many-to-many relationship, one entity has a relationship with many entities of another type, and those entities have a relationship with many entities of the first entity type. For example, a student can attend many courses, and a course can have numerous students attending it.

Suppose we have a service that stores student and course entities in a database. Each student entity contains the courses the student has attended. Similarly, each course entity contains a list of students that have attended the course. We have a many-to-many relationship where one student can attend multiple courses, and multiple students can attend one course. This means an additional mapping table, `studentcourse`, must be created. This new table maps a particular student to a particular course. Below is the many-to-many relationship implemented with JPA:

```
@Entity
@Table(name = "students")
class Student {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // Other Student fields...

  @JoinTable(
    name = "studentcourse",
    joinColumns = @JoinColumn(name = "studentid"),
    inverseJoinColumns = @JoinColumn(name = "courseid")
  )
  @ManyToMany
  private List<Course> attendedCourses;
}

@Entity
@Table(name = "courses")
class Course {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // Other Course fields...

  @ManyToMany(mappedBy = "attendedCourses")
  private List<Student> students;
}
```

The ORM creates the `students` and `courses` tables in addition to the `studentcourse` mapping table:

```
CREATE TABLE studentcourse (
  studentid BIGINT,
  courseid BIGINT,
  CONSTRAINT fkstudent FOREIGN KEY (studentid)
    REFERENCES students(id),
  CONSTRAINT fkorder FOREIGN KEY (courseid)
    REFERENCES courses(id)
);
```

Below is an example SQL query that the ORM executes to fetch attended courses for the user identified with id 123:

```
SELECT s.id, c.id, ...
FROM students s
LEFT JOIN studentcourse sc ON s.id = sc.studentid
LEFT JOIN courses c ON c.id = sc.courseid
WHERE s.id = 123;
```

Below is an example SQL query that the ORM executes to fetch students for the course identified with id 123:

```
SELECT c.id, s.id
FROM courses c
LEFT JOIN studentcourse sc ON c.id = sc.courseid
LEFT JOIN students s ON s.id = sc.studentid
WHERE c.id = 123;
```

In real-life scenarios, we don't necessarily have to or should implement many-to-many database relations inside a single microservice. For example, the above service that handles students and courses is against the *single responsibility principle*. We should create a separate microservice for students and a separate microservice for courses. Then there won't be many-to-many relationships between database tables in a single microservice.

Use Parameterized SQL Statements Principle

> *If you are not using an ORM for database access, use parameterized SQL statements to prevent potential SQL injection attacks.*

Let's use Node.js and the *mysql* NPM library for parameterized SQL examples. First, let's insert data to the `salesitems` table:

```

```

```
// Create a connection...

connection.query(
  `INSERT INTO salesitems (name, price)
   VALUES (?, ?)`,
  ['Sample sales item', 10]
);
```

The question marks (?) are placeholders for parameters in a parameterized SQL query. The second argument to the `query` method contains the parameter values. When a database engine receives a parameterized query, it will replace the placeholders in the query with the supplied parameter values.

Next, we can update a row in the `salesitems` table. The below example changes the price of the sales item with id 123 to 20:

```
connection.query('UPDATE salesitems SET price = ? WHERE id = ?',
                 [20, 123]);
```

Let's execute a SELECT statement to get sales items with their price over 20:

```
connection.query(
  'SELECT id, name, price FROM salesitems WHERE price >= ?',
  [20]
);
```

In an SQL SELECT statement, you cannot use parameters everywhere. You can use them as value placeholders in the WHERE clause. If you want to use user-supplied data in other parts of an SQL SELECT statement, you need to use string concatenation. You should not concatenate user-supplied data without sanitation because that would open up possibilities for SQL injection attacks. Let's say you allow the microservice client to specify a sorting column:

```
const sortColumn = // Unvalidated data got from client
const sqlQuery =
  'SELECT id, name, price FROM salesitems ORDER BY ' +
  connection.escapeId(sortColumn);

connection.query(sqlQuery);
```

As shown above, you need to escape the `sortColumn` value so that it contains only valid characters for a MySQL column name. If you need to get the sorting direction from the client, you should validate that value to be either `ASC` or `DESC`. In the below example, we assume that a `validateSortDirection` function exists:

```
const sortColumn = // Unvalidated data got from client
const sortDirection = // Unvalidated data got from client

// throws if invalid sorting direction
const validatedSortDirection =
  validateSortDirection(sortDirection);

const sqlQuery = `
  SELECT id, name, price
  FROM salesitems
  ORDER BY
  ${connection.escapeId(sortColumn)}
  ${validatedSortDirection}
`;

connection.query(sqlQuery);
```

When you get values for a MySQL query's `LIMIT` clause from a client, you must validate that those values are integers and in a valid range. Don't allow the client to supply random, very large values. In the example below, we assume that two validation functions exist: `validateRowOffset` and `validateRowCount`. The validation functions will throw if validation fails.

```
const rowOffset = // Unvalidated data got from client
const rowCount = // Unvalidated data got from client

const validatedRowOffset = validateRowOffset(rowOffset);
const validatedRowCount = validateRowCount(rowCount);

const sqlQuery = `
  SELECT id, name, price
  FROM salesitems
  LIMIT ${validatedRowOffset}, ${validatedRowCount}
`;

connection.query(sqlQuery);
```

When you get a list of wanted column names from a client, you must validate that each of them is a valid column identifier:

```
const columnNames = // Unvalidated data got from client

const escapedColumnNames =
  columnNames.map(columnName => connection.escapedId(columnName));

const sqlQuery =
  `SELECT ${escapedColumnNames.join(', ')} FROM salesitems`;

connection.query(sqlQuery);
```

Normalization Rules

> *Apply normalization rules to your database design.*

Below are listed the three most basic normalization rules:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)

A database relation is often described as "normalized" if it meets the first, second, and third normal forms.

First Normal Form (1NF)

The first normal form requires that at every intersection of a row and column, a single value exists and never a list of values. When considering a sales item, the first normal form states that there cannot be two different price values in the `price` column or more than one name for the sales item in the `name` column. If you need multiple names for a sales item, you must establish a one-to-many relationship between a `SalesItem` entity and `SalesItemName` entities. What this means in practice is that you remove the `name` property from the `SalesItem` entity class and create a new `SalesItemName` entity class used to store sales items' names. Then you create a one-to-many mapping between a `SalesItem` entity and `SalesItemName` entities.

Second Normal Form (2NF)

The second normal form requires that each non-key column entirely depends on the primary key. Let's assume that we have the following columns in an `orderitems` table:

- orderid (primary key)
- productid (primary key)
- orderstate

The `orderstate` column only depends on the `orderid` column, not the entire primary key. The `orderstate` column is in the wrong table. It should, of course, be in the `orders` table.

Third Normal Form (3NF)

The third normal form requires that non-key columns are independent of each other.

Let's assume that we have the following columns in a `salesitems` table:

- id (primary key)
- name
- price
- category

- discount

Let's assume that the discount depends on the category. This table violates the third normal form because a non-key column, `discount`, depends on another non-key column, `category`. Column independence means that you can change any non-key column value without affecting any other column. If you changed the category, the discount would need to be changed accordingly, thus violating the third normal form rule.

The discount column should be moved to a new `categories` table with the following columns:

- id (primary key)
- name
- discount

Then we should update the `salesitems` table to contain the following columns:

- id (primary key)
- name
- price
- categoryid (a foreign key that references the `id` column in the `categories` table)

Document Database Principle

> *Use a document database in cases where complete documents (e.g., JSON objects) are typically stored and retrieved as a whole.*

Document databases, like MongoDB, are useful for storing complete documents. A document is usually a JSON object containing information in arrays and nested objects. Documents are stored as such, and a whole document will be fetched when queried.

Let's consider a microservice for sales items. Each sales item contains an id, name, price, image URLs, and user reviews.

Below is an example sales item as a JSON object:

```
{
  "id": "507f191e810c19729de860ea",
  "category": "Power tools",
  "name": "Sample sales item",
  "price": 10,
  "imageUrls": ["https://url-to-image-1...",
                "https://url-to-image-2..."],
```

```
  "averageRatingInStars": 5,
  "reviews": [
     {
       "reviewerName": "John Doe",
       "date": "2022-09-01",
       "ratingInStars": 5,
       "text": "Such a great product!"
     }
  ]
}
```

A document database usually has a size limit for a single document. Therefore, the above example does not store sales item images directly inside the document but only URLs to the images. Actual images are stored in another data store more suitable for storing images, like Amazon S3.

When creating a microservice for sales items, we can choose a document database because we usually store and access whole documents. When sales items are created, they are created as JSON objects of the above shape with the `reviews` array being empty. When a sales item is fetched, the whole document is retrieved from the database. When a client adds a review for a sales item, the sales item is fetched from the database. The new review is appended to the `reviews` array, a new average rating is calculated, and finally, the document is persisted with the modifications.

Below is an example of inserting one sales item to a MongoDB collection named `salesItems`. MongoDB uses the term *collection* instead of *table*. A MongoDB collection can store multiple documents.

```
db.salesItems.insertOne({
  category: "Power tools",
  name: "Sample sales item",
  price: 10,
  images: ["https://url-to-image-1...",
           "https://url-to-image-2..."],
  averageRatingInStars: null,
  reviews: []
})
```

You can find sales items for the *Power tools* category with the following query:

```
db.salesItems.find({ category: "Power tools" })
```

If clients are usually querying sales items by category, it is wise to create an index for that field:

```
// 1 means ascending index, -1 means descending index
db.salesItems.createIndex( { category: 1 } )
```

When a client wants to add a new review for a sales item, you first fetch the document for the sales item:

```
db.salesItems.find({ _id: ObjectId("507f191e810c19729de860ea") })
```

Then you calculate a new value for the `averageRatingInStars` field using the existing ratings and the new rating

and add the new review to the `reviews` array and then update the document with the following command:

```
db.salesItems.updateOne(
  { _id: ObjectId("507f191e810c19729de860ea")  },
  { averageRatingInStars: 5,
  $push: { reviews: {
    reviewerName: "John Doe",
    date: "2022-09-01",
    ratingInStars: 5,
    text: "Such a great product!"
  }}
)
```

Clients may want to retrieve sales items sorted descending by the average rating. For this reason, you might want to change the indexing to be the following:

```
db.salesItems.createIndex( { category: 1, averageStarCount: -1 } )
```

A client can issue, for example, a request to get the best-rated sales items in the *Power tools* category. This request can be fulfilled with the following query that utilizes the above-created index:

```
db.salesItems
  .find({ category: "Power tools" })
  .sort({ averageStarCount: -1 })
```

Key-Value Database Principle

> *Use a key-value database for fast real-time access to data stored by a key. Key-value stores usually store data in memory with a possibility for persistence.*

A simple use case for a key-value database is to use it as a cache for a relational database. For example, a microservice can store SQL query results from a relational database in the cache. *Redis* is a popular open-source key-value store. Let's have an example with JavaScript and Redis to cache an SQL query result. In the below example, we assume that the SQL query result is available as a JavaScript object:

```
redisClient.set(sqlQueryStatement,
                JSON.stringify(sqlQueryResult));
```

The cached SQL query result can be fetched from Redis:

```
const sqlQueryResultJson = redisClient.get(sqlQueryStatement);
```

With Redis, you can create key-value pairs that expire automatically after a specific time. This is a useful feature if you are using the key-value database as a cache. You may want the cached items to expire after a while.

In addition to plain strings, Redis also supports other data structures. For example, you can store a list, queue, or hash map for a key. If you store a queue in Redis, you can use it as a simple single-consumer message broker. Below is an example of producing a message to a topic in the message broker:

```
// RPUSH command (= right push) pushes a new message
// to the end of the list identified by key _topic_.
redisClient.rpush(topic, message);
```

Below is an example of consuming a message from a topic in the message broker:

```
// LPOP command (= left pop) pops a message from
// the beginning of the list identified by key _topic_
// The LPOP command removes the value from the list
const message = redisClient.lpop(topic);
```

Wide-Column Database Principle

> *Use a wide-column database when you know what queries you need to execute, and you want these queries to be fast.*

Table structures of a wide-column database are optimized for specific queries. With a wide-column database, storing duplicate data is okay to make the queries faster. Wide-column databases also scale horizontally well.

In this section, we use Apache Cassandra as an example wide-column database. Cassandra is a scalable multi-node database engine. In Cassandra, the data of a table is divided into partitions according to the table's partition key. A partition key is composed of one or more columns. Each partition is stored on a single Cassandra node. You can think that Cassandra is a key-value store where the key is the partition key, and the value is another "nested" table. The rows in the "nested" table are uniquely identified by clustering columns sorted by default in ascending order. The sort order can be changed to descending if wanted.

The partition key and the clustering columns form the table's primary key. The primary key uniquely identifies a row. Let's have an example table that is used to store hotels near a particular point of interest (POI):

```
CREATE TABLE hotels_by_poi (
  poi_name text,
  hotel_distance_in_meters_from_poi int,
  hotel_id uuid,
  hotel_name text,
```

```
  hotel_address text,
  PRIMARY KEY (poi_name, hotel_distance_in_metersfrom_poi, hotel_id)
);
```

In the above example, the primary key consists of three columns. The first column (`poi_name`) is always the partition key. The partition key must be given in a query. Otherwise, the query will be slow because Cassandra must perform a full table scan because it does not know which node data is located. When the partition key is given in a SELECT statement's WHERE clause, Cassandra can find the appropriate node where the data for that particular partition resides. The two other primary key columns, `hotel_distance_in_meters_from_poi` and `hotel_id` are the clustering columns. They define the order and uniqueness of the rows in the "nested" table.

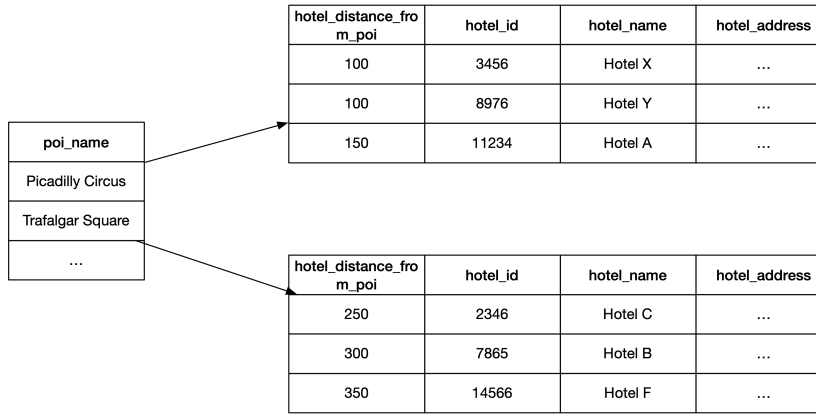

Figure 8.1 hotels_by_poi Table

The above figure shows that when you give a partition key value (`poi_name`) you have access to the respective "nested" table where rows are ordered first by the `hotel_distance_in_meters_from_poi` (ascending) and second by the `hotel_id` (ascending).

Now it is easy for a hotel room booking client to ask the server to execute a query to find hotels near a POI given by a user. The following query will return the first 15 hotels nearest to *Piccadilly Circus* POI:

```
SELECT
  hotel_distance_in_meters_from_poi,
  hotel_id,
  hotel_name,
  hotel_address
FROM hotels_by_poi
WHERE poi_name = 'Piccadilly Circus'
LIMIT 15
```

When a user selects a particular hotel from the result of the above query, the client can request the execution of another query to fetch information about the selected hotel. The user wants to see other POIs near the selected hotel. For that query, we should create another table:

```
CREATE TABLE pois_by_hotel_id (
  hotel_id uuid,
  poi_distance_in_meters_from_hotel int,
  poi_id uuid,
  poi_name text,
  poi_address text,
  PRIMARY KEY (hotel_id, poi_distance_in_meters_from_hotel, poi_id)
);
```

Now a client can request the server to execute a query to fetch the nearest 20 POIs for a selected hotel. (hotel with id c5a49cb0-8d98-47e3-8767-c30bc075e529):

```
SELECT
  poi_distance_in_meters_from_hotel,
  poi_id,
  poi_name,
  poi_address
FROM pois_by_hotel_id
WHERE hotel_id = c5a49cb0-8d98-47e3-8767-c30bc075e529
LIMIT 20
```

In a real-life scenario, a user wants to search for hotels near a particular POI for a selected period of time. The server should respond with the nearest hotels having free rooms for the selected period. For that kind of query, we create an additional table for storing hotel room availability:

```
CREATE TABLE availability_by_hotel_id (
  hotel_id uuid,
  accommodation_date date,
  available_room_count counter,
  PRIMARY KEY (hotel_id, accommodation_date)
);
```

The above table is updated whenever a room for a specific day is booked or a booking for a room is canceled. The `available_room_count` column value is either decremented or incremented by one in the update procedure.

Let's say that the following query has been executed:

```
SELECT
  hotel_distance_in_meters_from_poi,
  hotel_id,
  hotel_name,
  hotel_address
FROM hotels_by_poi
WHERE poi_name = 'Piccadilly Circus'
LIMIT 30
```

Next, we should find hotels from the result of 30 hotels that have available rooms between the 1st of September 2022 and 3rd of September 2022. We cannot use joins in Cassandra, but we can execute

the following query where we specifically list the hotel ids returned by the above query:

```
SELECT hotel_id, MIN(available_room_count)
FROM availability_by_hotel_id
WHERE hotel_id IN (List the 30 hotel_ids here...) AND
      accommodation_date >= '2022-09-01' AND
      accommodation_date <= '2022-09-03'
GROUP BY hotel_id
LIMIT 15
```

As a result of the above query, we have a list of a maximum of 15 hotels for which the minimum available room count is listed. We can return a list of those max 15 hotels where the minimum available room count is one or more to the user.

If Cassandra's CQL language supported the HAVING clause, which it does not currently support, we could have issued the following query to get what we wanted:

```
SELECT hotel_id, MIN(available_room_count)
FROM availability_by_hotel_id
WHERE hotel_id IN (List the 30 hotel_ids here...) AND
      accommodation_date >= '2022-09-01' AND
      accommodation_date <= '2022-09-03'
GROUP BY hotel_id
HAVING MIN(available_room_count) >= 1
LIMIT 15
```

A wide-column database is also useful in storing time-series data from IoT devices and sensors. Below is a table definition for storing measurement data in a telecom network analytics system:

```
CREATE TABLE measurements (
  measure_name text,
  dimension_name text,
  aggregation_period text,
  measure_timestamp timestamp,
  measure_value double,
  dimension_value text,
  PRIMARY KEY ((measure_name, dimension_name, aggregation_period),
               measure_timestamp,
               measure_value,
               dimension_value)
) WITH CLUSTERING ORDER BY (
  measure_timestamp DESC,
  measure_value DESC
  dimension_value ASC
);
```

In the above table, we have defined a *compound partition key* containing three columns: `measure_name`, `dimension_name`, and `aggregation_period`. Columns for a compound partition key are given in parentheses.

Suppose we have implemented a client that visualizes measurements. In the client, a user can first

choose what counter/KPI (= measure name) to visualize, then select a dimension and aggregation period. Let's say that the user wants to see *dropped_call_percentage* for *cells* calculated for a one-minute period at 2022-02-03 16:00. The following kind of

query can be executed:

```
SELECT measure_value, dimension_value
FROM measurements
WHERE measure_name = 'dropped_call_percentage' AND
      dimension_name = 'cell' AND
      aggregation_period = '1min' AND
      measureTimestamp = '2022-02-03T16:00+0000'
LIMIT 50;
```

The above query returns the top 50 cells where the dropped call percentage is highest for the given minute.

We can create another table to hold measurements for a selected dimension value, e.g., for a particular cell id. This table can be used to drill down to a particular dimension and see measure values in the history.

```
CREATE TABLE measurements_by_dimension (
  measure_name text,
  dimension_name text,
  aggregation_period text,
  dimension_value text,
  measure_timestamp timestamp,
  measure_value double,
  PRIMARY KEY ((measure_name,
                dimension_name,
                aggregation_period,
                dimension_value), measure_timestamp)
) WITH CLUSTERING ORDER BY (measureTimestamp DESC);
```

The below query will return dropped call percentage values for the last 30 minutes for the cell identified by *cell id* 3000:

```
SELECT measure_value, measureTimestamp
FROM measurements_by_dimension
WHERE measure_name = 'dropped_call_percentage' AND
      dimension_name = 'cell' AND
      aggregation_period = '1min' AND
      dimension_value = '3000'
LIMIT 30;
```

Search Engine Principle

Use a search engine if you have free-form text data that users should be able to query.

A search engine (like Elasticsearch, for example) is useful for storing information like log entries collected from microservices. You typically want to search the collected log data by the text in the log messages.

It is not necessary to use a search engine when you need to search for text data. Other databases, both document and relational, have a special index type that can index free-form text data in a column. Considering the earlier example with MongoDB, we might want a client to be able to search sales items by the text in the sales item's name. We don't need to store sales items in a search engine database. We can continue storing them in a document database (MongoDB) and introduce a *text* type index for the `name` field. That index can be created with the following MongoDB command:

```
db.salesItems.createIndex( { name: "text" } )
```

Concurrent Programming Principles

This chapter presents the following concurrent programming principles:

- Threading principle
- Thread safety principle

Threading Principle

> *Modern cloud-native microservices should primarily scale out by adding more processes, not scale up by adding more threads. Use threading only when it is needed or is a good optimization.*

When developing modern cloud-native software, microservices should be stateless and automatically scale horizontally (scaling out and in via adding and removing processes). The role of threading in modern cloud-native microservices is not as prominent as it was earlier when software consisted of monoliths running on bare metal servers, mainly capable of scaling up or down. Nowadays, you should use threading if it is a good optimization or otherwise needed.

Suppose we have a software system with an event-driven architecture. Multiple microservices communicate with each other using asynchronous messaging. Each microservice instance has only a single thread that consumes messages from a message broker and then processes them. If the message broker's message queue for a microservice starts growing too long, the microservice should scale out by adding a new instance. When the load for the microservice diminishes, it can scale in by removing an instance. There is no need to use threading at all.

We could use threading in the data exporter microservice if the input consumer and the output producer were synchronous. The reason for threading is optimization. If we had everything in a single thread and the microservice was performing network I/O (either input or output-related),

the microservice would have nothing to execute because it is waiting for some network I/O to complete. Using threads, we can optimize the execution of the microservice so that it potentially has something to do when waiting for an I/O operation to complete.

Many modern input consumers and output producers are available as asynchronous implementations. If we use an asynchronous consumer and producer in the data exporter microservice, we can eliminate threading because network I/O will not block the execution of the main thread anymore. As a rule of thumb, consider using asynchronous code first, and if it is not possible or feasible, only then consider threading.

You might need a microservice to execute housekeeping tasks on a specific schedule in the background. Instead of using threading and implementing the housekeeping functionality in the microservice, consider implementing it in a separate microservice to ensure that the *single responsibility principle* is followed. You can configure the housekeeping microservice to be run at regular intervals using a Kubernetes CronJob, for example.

Threading also brings complexity to a microservice because the microservice must ensure thread safety. You will be in big trouble if you forget to implement thread safety. Threading and synchronization-related bugs are hard to find. Thread safety is a topic that is discussed later in this chapter. Threading also brings complexity to deploying a microservice because the number of vCPUs requested by the microservice can depend on the thread count.

Parallel Algorithms

Parallel algorithms are similar to threading. With parallel algorithms, it is a question about implicit threading instead of explicit threading. Threads are created behind the scenes to enable an algorithm to execute in parallel on some data set. A parallel algorithm is usually something you don't necessarily need in a cloud-native microservice. You can often run algorithms without parallelization and instead scale out on demand.

Many languages offer parallel algorithms. In Java, you can perform parallel operations with a parallel stream created with the `parallelStream` method:

```
final var numbers = List.of(1, 2, 3, 4);
numbers.parallelStream().forEach(number ->
  System.out.println(number + " " +
                     Thread.currentThread().getName())
);
```

The output of the above code could be, for example:

```
3 ForkJoinPool.commonPool-worker-2
2 ForkJoinPool.commonPool-worker-1
1 ForkJoinPool.commonPool-worker-3
4 main
```

The output will differ on each run. Usually, the parallel executor creates the same amount of threads as there are CPU cores available. This means that you can scale your microservice up by requesting more CPU cores. In many languages, you can control how many CPU cores the parallel algorithm should use. You can, for example, configure that a parallel algorithm should use the number of available CPU cores minus two if you have two threads dedicated to some other processing. In C++20, you cannot control the number of threads for a parallel algorithm, but an improvement is coming in a future C++ release.

Below is the same example as above written in C++:

```
#include <algorithm>
#include <execution>
#include <thread>
#include <iostream>

std::vector<int> numbers{1, 2, 4, 5};

std::for_each(std::execution::par,
              numbers.cbegin(),
              numbers.cend(),
              [](const auto number)
              {
                std::cout << number
                          << " "
                          << std::this_thread::get_id()
                          << "\n";
              });
```

Thread Safety Principle

> *If you are using threads, you must ensure thread safety. Thread safety means that only one thread can access shared data simultaneously to avoid race conditions.*

Do not assume thread safety if you use a data structure or library. You must consult the documentation to see whether thread safety is guaranteed. If thread safety is not mentioned in the documentation, it can't be assumed. The best way to communicate thread safety to developers is to name things so that thread safety is explicit. For example, you could create a thread-safe collection library and have a class named `ThreadSafeLinkedList` to indicate the class is thread-safe. Another common word used to indicate thread safety is *concurrent*, e.g., the `ConcurrentHashMap` class in Java.

There are several ways to ensure thread safety:

- Synchronization directive
- Atomic variables
- Concurrent collections
- Mutexes
- Spin locks

The subsequent sections describe each of the above techniques in more detail.

Synchronization Directive

In some languages, you can use a specific directive to indicate that a particular piece of code is synchronized, meaning that only one thread can execute that piece of code at a time.

Java offers the `synchronized` keyword that can be used in the following ways:

- Synchronized instance method
- Synchronized static method
- Synchronized code block

```
public synchronized void doSomething() {
  // Only one thread can execute this at the same time
}

public static synchronized void doSomething() {
  // Only one thread can execute this at the same time
}

public void doSomething() {
  // ...

  synchronized (this) {
    // Only one thread can execute this at the same time
  }

  // ...
}
```

Atomic Variables

If you have some data shared between threads and that data is just a simple primitive variable, like a boolean or integer, you can use an atomic variable to guarantee thread safety and don't need to use any additional synchronization technique. Atomic variable reads and updates

are guaranteed to be atomic, so there is no possibility for a race condition between two threads. Some atomic variable implementations use locks. Consult the language's documentation to see if a certain atomic type is guaranteed to be lock-free.

In C++, you can create a thread-safe counter using an atomic variable:

ThreadSafeCounter.h

```
#include <atomic>

class ThreadSafeCounter
{
public:
  ThreadSafeCounter() = default;

  void increment()
  {
    ++m_counter;
  }

  uint32_t getValue() const
  {
    return m_counter.load();
  }

private:
  std::atomic<uint64_t> m_counter{0U};
}
```

Concurrent Collections

Concurrent collections can be used by multiple threads without any additional synchronization. Java offers several concurrent collections in the *java.util.concurrent* package.

C++ does not offer concurrent collections in its standard library. You can create a concurrent, i.e., a thread-safe collection using the *decorator pattern* by adding needed synchronization (locking) to an existing collection class. Below is a partial example of a thread-safe vector created in C++:

ThreadSafeVector.h

```
#include <vector>

template <typename T>
class ThreadSafeVector
{
public:
  explicit ThreadSafeVector(std::vector<T>&& vector):
    m_vector{std::move(vector)}
  {}

  void pushBack(const T& value)
  {
    // Lock using a mutex or spin lock

    m_vector.push_back(value);
```

```
    // Unlock
  }

  // Implement other methods with locking...

private:
  std::vector<T> m_vector;
};
```

Mutexes

A mutex is a synchronization primitive that can protect shared data from being simultaneously accessed by multiple threads. A mutex is usually implemented as a class with `lock` and `unlock` methods. The locking only succeeds by one thread at a time. Another thread can lock the mutex only after the previous thread has unlocked the mutex. The `lock` method waits until the mutex becomes available for locking.

Let's implement the `pushBack` method using a mutex:

ThreadSafeVector.h

```
#include <mutex>
#include <vector>

template <typename T>
class ThreadSafeVector
{
public:
  explicit ThreadSafeVector(std::vector<T>&& vector):
    m_vector{std::move(vector)}
  {}

  void pushBack(const T& value)
  {
    m_mutex.lock();
    m_vector.push_back(value);
    m_mutex.unlock();
  }

  // Implement other methods ...

private:
  std::vector<T> m_vector;
  std::mutex m_mutex;
};
```

Mutexes are not usually used directly because there exists the risk that a mutex is forgotten to be unlocked. Instead of using the plain `std::mutex` class, you can use a mutex with the `std::scoped_lock` class. The `std::scoped_lock` class wraps a mutex instance. It will lock the wrapped mutex on construction and unlock the mutex on destruction. In this way, you cannot forget to unlock a locked mutex. The mutex will be locked for the scope of the scoped lock variable. Below is the above example modified to use a scoped lock:

ThreadSafeVector.h

```
#include <mutex>
#include <vector>

template <typename T>
class ThreadSafeVector
{
public:
  explicit ThreadSafeVector(std::vector<T>&& vector):
    m_vector{std::move(vector)}
  {}

  void pushBack(const T& value)
  {
    const std::scoped_lock scopedLock{m_mutex};
    m_vector.push_back(value);
  }

  // Implement other methods ...

private:
  std::vector<T> m_vector;
  std::mutex m_mutex;
};
```

Spinlocks

A spinlock is a lock that causes a thread trying to acquire it to simply wait in a loop (spinning the loop) while repeatedly checking whether the lock has become available. Since the thread remains active but is not performing a useful task, using a spinlock is busy waiting. You can avoid some of the overhead of thread context switches using a spinlock. Spinlocks are an effective way of locking if the locking periods are short.

Let's implement a spinlock using C++:

Spinlock.h

```
#include <atomic>
#include <thread>

#include <boost/core/noncopyable.hpp>

class Spinlock : public boost::noncopyable
{
public:
  void lock()
  {
    while (true) {
      const bool wasLocked =
        m_isLocked.test_and_set(std::memory_order_acquire);

      if (!wasLocked) {
        // Is now locked
        return;
      }
```

```
      // Wait for the lock to be released
      while (m_isLocked.test(std::memory_order_relaxed)) {
        // Prioritize other threads
        std::this_thread::yield();
      }
    }
  }

  void unlock()
  {
    m_isLocked.clear(std::memory_order_release);
  }

private:
  std::atomic_flag m_isLocked = ATOMIC_FLAG_INIT;
};
```

In the above implementation, we use the `std::atomic_flag` class because it guarantees a lock-free implementation across all C++ compilers. We also use a non-default memory ordering to allow the compiler to emit more efficient code.

Now we can re-implement the `ThreadSafeVector` class using a spinlock instead of a mutex:

ThreadSafeVector.h

```
#include <vector>
#include "Spinlock.h"

template <typename T>
class ThreadSafeVector
{
public:
  explicit ThreadSafeVector(std::vector<T>&& vector):
    m_vector{std::move(vector)}
  {}

  void pushBack(const T& value)
  {
    m_spinlock.lock();
    m_vector.push_back(value);
    m_spinlock.unlock();
  }

  // Implement other methods ...

private:
  std::vector<T> m_vector;
  Spinlock m_spinlock;
};
```

Similar to mutexes, we should not use raw spinlocks in our code but use a scoped lock. Below is an implementation of a generic `ScopedLock` class that handles the locking and unlocking of a lockable object:

ScopedLock.h

```
#include <concepts>

#include <boost/core/noncopyable.hpp>

template<typename T>
concept Lockable = requires(T a)
{
    { a.lock() } -> std::convertible_to<void>;
    { a.unlock() } -> std::convertible_to<void>;
};

template <Lockable L>
class ScopedLock : public boost::noncopyable
{
public:
  explicit ScopedLock(L& lockable):
    m_lockable{lockable}
  {
    m_lockable.lock();
  }

  ~ScopedLock()
  {
    m_lockable.unlock();
  }

private:
  L& m_lockable;
};
```

Let's change the `ThreadSafeVector` class to use a scoped lock:

ThreadSafeVector.h

```
#include <vector>
#include "Scopedlock.h"
#include "Spinlock.h"

template <typename T>
class ThreadSafeVector
{
public:
  explicit ThreadSafeVector(std::vector<T>&& vector):
    m_vector{std::move(vector)}
  {}

  void pushBack(const T& value)
  {
    const ScopedLock scopedLock{m_spinlock};
    m_vector.push_back(value);
  }

  // Implement other methods ...

private:
  std::vector<T> m_vector;
  Spinlock m_spinlock;
};
```

Teamwork Principles

This chapter presents teamwork principles. The following principles are described:

- Use an agile framework principle
- Define the done principle
- You write code for other people principle
- Avoid technical debt principle
- Software component documentation principle
- Code review principle
- Uniform code formatting principle
- Highly concurrent development principle
- Pair programming principle
- Well-defined development team roles principle

Use Agile Framework Principle

> *Using an agile framework can bring numerous benefits to an organization, including an increase in productivity, improvements in quality, faster time-to-market, and better employee satisfaction.*

The above statements come from customer stories https://scaledagile.com/insights-customer-stories of some companies having adopted Scaled Agile Framework (SAFe) (https://www.scaledagileframework.com/).

An agile framework describes a standardized way of developing software, which is essential, especially in large organizations. In today's work environments, people change jobs frequently, and teams tend to change often, which can lead to a situation where there is no common understanding

of the way of working unless a particular agile framework is used. An agile framework establishes a clear division of responsibilities, and everyone can focus on what they do best.

In the *SAFe*, for example, during a program increment (PI) planning, development teams plan features for the next PI (consisting of 4 iterations, two weeks per iteration, a total of 8 weeks). In the PI planning, teams split features into user stories and see which features fit in the PI. Planned user stories will be assigned story points (measured in person days, for example), and stories will be placed into iterations. This planning phase results in a plan the team should follow in the PI. Junior SAFe practitioners can make mistakes like underestimating the work needed to complete a user story. But this is a self-correcting issue. When teams and individuals develop, they will better estimate the needed work amount, and plans become more solid. Teams and developers learn that they must make all work visible. For example, reserve time to learn new things, like a programming language or framework, and reserve time for refactoring. It is very satisfying to keep the planned schedule and sometimes even complete work early. This will make you feel like a true professional and is a boost to self-esteem.

Define the Done Principle

> *For user stories and features, define what **done** means.*

In the most optimal situation, development teams have a shared understanding of what is needed to declare a *user story* or *feature* done. When having a common definition of done, each development team can ensure consistent results and quality.

When considering a user story, at least the following requirements for a done user story can be defined:

- Source code is committed to a source code repository
- Source code is reviewed
- Static code analysis is performed (No blocker/critical/major issues)
- Unit test coverage is at least X%
- CI/CD pipeline is passing
- No 3rd party software vulnerabilities

The product owner's (PO) role in a team is to accept a user story as done. Some of the above-mentioned requirements can be automatically checked. For example, the static code analysis should be part of every CI/CD pipeline and can also check the unit test coverage automatically. If static code analysis does not pass or the unit test coverage is not acceptable, the CI/CD pipeline does not pass.

Some additional requirements for done-ness should be defined when considering a feature because features can be delivered to customers. Below is a list of some requirements for a done feature:

- Architectural design documentation is updated
- Integration tests are added/updated
- End-to-end tests are added/updated if needed
- Non-functional testing is done
- User documentation is ready
- Threat modeling is done, and threat countermeasures (security features) are implemented

To complete all the needed done-ness requirements, development teams can use tooling that helps them remember what needs to be done. For example, when creating a new user story in a tool like Jira, an existing prototype story could be cloned (or a template used). The prototype or template story should contain tasks that must be completed before a user story can be approved.

You Write Code for Other People Principle

> *You write code for other people and your future self.*

Situations where you work alone with a piece of software are relatively rare. You cannot predict what will happen in the future. There might be someone else responsible for the code you once wrote. And there are cases when you work with some code for some time and then, maybe after several years, need to return to that code. For these reasons, writing clean code that is easy to read and understand by others and yourself in the future is essential. Remember that code is not written for a computer only but also for people. People should be able to read and comprehend code easily. Remember that at its best, code reads like beautiful prose!

Avoid Technical Debt Principle

The most common practices for avoiding technical debt are the following:

- The architecture team should design the high-level architecture (Each team should have a representative in the architecture team. Usually, it is the technical lead of the team)
- Development teams should perform object-oriented design first, and only after that proceed with implementation
- Conduct object-oriented design within the team with relevant senior and junior developers involved

- Don't take the newest 3rd party software immediately into use, instead use mature 3rd party software that has an established position in the market
- Design for easily replacing a 3rd party software component with another 3rd party component.
- Design for scalability (for future load)
- Design for extension: new functionality is placed in new classes instead of modifying existing classes
- Utilize a plugin architecture (possibility to create plugins to add new functionality later)
- Reserve time for refactoring

The top reasons for technical debt are the following:

- Using niche technologies or brand-new technologies that are immature
- Not making software scalable for future processing needs
- When it is not relatively easy to replace a 3rd party software component (E.g., using custom SQL syntax does not allow changing the database, not using the *adapter pattern* with 3rd party libraries)
- Not reviewing the architecture
- Not doing any object-oriented design before starting coding
- Not engaging senior enough developers in the OOD phase
- Not understanding and using relevant design principles and patterns
    - Not programming against interfaces
    - Not easy to change a dependency (DI missing)
    - No facades
- Not reviewing code changes
- Not reserving time for refactoring
- Too small work effort estimates
- Time pressure from management
- Management does not understand the value of refactoring
- Postponing refactoring to a time point that never comes
- Forgetting to refactor (at least store the needed refactoring work items in a TODO.MD file in the source code repository)
- No unit tests, harder to refactor
- Duplicate code
- Not conducting the boy scout rule
- Laziness (using what comes first in mind or constantly trying to find the easiest and quickest way to do things)

Software Component Documentation Principle

> *Each software component needs to be documented. The main idea behind documenting is quickly onboarding new people to development work.*

It is crucial that setting up a development environment for a software component is well-documented and as easy as possible. Another important thing is to let people easily understand the problem domain the software component tries to solve. Also, the object-oriented design of the software component should be documented.

Software component documentation should reside in the same source code repository where the source code is. The recommended way is to use a README.MD file in the root directory of the source code repository for documentation in Markdown format. You can split the documentation into multiple files and store additional files in the *docs* directory of the source code repository.

Below is an example table of contents that can be used when documenting a software component:

- Short description of the software component and its purpose
- Feature list
- OOD diagrams describing different subdomains and classes in each subdomain
    - Explanation of the design (if needed)
- API documentation (for libraries)
- Implementation-related documentation
    - Error handling mechanism
    - Special algorithms used
    - Performance considerations
- Instructions for setting up a development environment
    - The easiest way to set up a development environment is to use a development container, a concept supported by the Visual Studio Code editor. The benefit of using a development container is that you don't have to install development tools locally, and there is no risk of using the wrong versions of the development tools
- Instructions for building the software locally
- Instructions for running unit tests locally
- Instructions for running integration tests locally
- Instructions for deploying to a test environment
- Configuration
    - Environment variables
    - Configuration files
    - Secrets

Code Review Principle

> *In a code review, focus on issues a machine cannot find for you.*

Before reviewing code, a static code analysis should be performed to find any issues a machine can find. The actual code review should focus on issues that static code analyzers cannot find. You cannot review your own code. At least one of the reviewers should be in a senior or lead role. Things to focus on in a code review are presented in the subsequent sections.

Focus on Object-Oriented Design

Before starting coding, it is recommended to design the software: define subdomains, needed interfaces, and classes. The product of this initial design phase should be committed to the source code repository and reviewed before starting coding. In this way, it is easier to correct design flaws early. Fixing design flaws in a later phase might require significant effort or even a rewrite of the existing code. At least one senior developer should participate in the design.

If a design flaw or flaws are encountered in a review, but there is no time for an immediate fix. A refactoring user story or stories should be added to the team's backlog.

Focus on Proper and Uniform Naming

One thing that static code analysis tools can only partially do is ensure proper and uniform naming of things, like classes, functions, and variables. Naming is where the focus should be put in a code review. Renaming things is a very straightforward and fast refactoring task that can be performed automatically by modern IDEs.

Don't Focus on Premature Optimization

Do not focus on optimization in regular code reviews. Optimization is usually performed on a need basis after the code is ready and the performance is first measured. Focus on optimization-related issues only when the commit you are reviewing is dedicated to optimizing something.

Detect Possible Malicious Code

Code reviewers must verify that the committed code does not contain malicious code.

Uniform Code Formatting Principle

> *In a software development team, you must decide on common rules for formatting source code.*

Consistent code formatting is vital because if team members have different source code formatting rules, one team member's small change to a file can reformat the whole file using his/hers formatting rules, which can cause another developer to face a major merge conflict that slows down the development process. Always agree on common source code formatting rules and preferably use a tool like *Prettier* to enforce the formatting rules. If no automatic formatting tool is available, you can create source code formatting rules for IDEs used by team members and store those rules in the source code repository.

Highly Concurrent Development Principle

> *Each team member can work with some piece of code. No one should be waiting long for someone else's work to finish.*

Concurrent development is enabled when different people modify different source code files. When several people need to alter the same files, it can cause merge conflicts. These merge conflicts cause extra work because they often must be resolved manually. This manual work can be slow, and it is error-prone. The best thing is to avoid merge conflicts as much as possible. This can be achieved in the ways described in the following sections.

Dedicated Microservices and Microlibraries

Microservices are small by nature, and it is possible to assign the responsibility of a microservice to a single team member. This team member can proceed with the microservice with full velocity, and rest assured that no one else is modifying the codebase. The same goes for libraries. You should create small microlibraries (= libraries with a single responsibility) and assign the responsibility of developing a microlibrary to a single person.

Dedicated Domains

Sometimes it is impossible to assign a single microservice or library to a single developer. It could be because the microservice or library is relatively large and it is not feasible to split it into multiple microservices or libraries. In those cases, the microservice or library should be divided into several subdomains by conducting domain-driven design. Each source code directory reflects a different

subdomain. It is then possible to assign the responsibility of a single subdomain to a single person. The assignment of subdomains should not be fixed but can and should change as time goes by. To distribute the knowledge of different domains, it is advisable to rotate the responsibilities amongst the team members. Let's say you have a team of three developers developing a data exporter microservice consisting of three subdomains: input, transformer, and output. The team can implement the microservice by assigning the responsibility of a single domain to a single developer. Now all developers can proceed highly independently and concurrently with the implementation. In the early phase, they must define interfacing between the different subdomains.

In the future, when new features are developed, team members can take responsibility for other domains to spread knowledge about the microservice in the team.

Follow Open-Closed Principle

Sometimes you might face a situation where a single subdomain is so large that you need multiple developers. This, of course, should be a relatively rare case. When several developers modify source code files belonging to the same subdomain (i.e., in the same directory), merge conflicts may arise. This is the case, especially when existing source code files are modified. But when developers follow the *open-closed principle*, they should not change existing classes (source code files) but rather implement new functionality in new classes (source code files). Using the *open-closed principle* enables developers to develop more concurrently because they primarily work with different source code files, making merge conflicts rare or at least less frequent.

Pair Programming Principle

> *Pair programming helps produce better quality software with better design, less technical debt, better tests, and fewer bugs.*

Pair programming is something some developers like, and other developers hate. So it is not a one-fits-all solution. It is not take it or leave it, either. You can have a team where some developers program in pairs and others don’t. Also, people's opinions about pair programming can be prejudiced. Perhaps, they have never done pair programming, so how do they know if they like it or not? It is also true that choosing the right partner to pair with can mean a lot. Some pairs have better chemistry than other pairs.

Does pair programming just increase development costs? What benefits pair programming brings?

I see pair programming as valuable, especially in situations where a junior developer pairs with a more senior developer, and in this way, the junior developer is onboarded much faster. He can "learn from the best". Pair programming can improve software design because there is always at

least two persons' view of the design. Bugs can be found easier and usually in an earlier phase (four eyes compared to two eyes only). So, even if pair programming can add some cost, it usually results in software with better quality: better design, less technical debt, better tests, and fewer bugs.

Well-Defined Development Team Roles Principle

A software development team should have a well-defined role for each team member.

A software development team does not function optimally if everyone is doing everything or if it is expected that anyone can do anything. No one is a jack of all trades. A team achieves the best results when it has specialists targeted for different tasks. Team members need to have focus areas they like to work with and where they can excel. When you are a specialist in some area, you can complete tasks belonging to that area faster and with better quality.

Below is a list of needed roles for a development team:

- Product owner (PO)
- Scrum master (SM)
- Software developer (junior/senior/lead)
- Test automation developer
- DevOps engineer
- UI designer (if the team develops UI software components)

Let's discuss each role's responsibilities in detail in the following sections.

Product Owner

The product owner (PO) acts as an interface between the development team and the business, which usually means product management. The PO is responsible for defining user stories and prioritizing the team backlog together with the team. The PO role is not usually a full-time role.

Scrum Master

A scrum master (SM) is a servant leader and a coach for the development team. The scrum master ensures that relevant agile practices and the agile process are followed. They educate the team in agile practices. If the team has a line manager, the line manager can serve as the scrum master, but any team member can be the scrum master.

Software Developer

A software developer is responsible for designing, implementing, and testing software (including unit and, in most cases, integration testing). A software developer is usually focused on one or two programming languages and a couple of technical frameworks. Typically, software developers are divided into the following categories:

- Backend developers
- Frontend developers
- Full-stack developers
- Mobile developers
- Embedded developers

A backend developer develops microservices, like APIs, running in the backend. A frontend developer develops web clients. Typically, a frontend developer uses JavaScript or TypeScript, React/Angular/Vue, HTML and CSS. A full-stack developer is a combination of a backend and frontend developer capable of developing backend microservices and frontend clients. A mobile developer develops software for mobile devices, like phones and tablets.

A team should have software developers at various seniority levels. Each team should have a lead developer with the best experience in the used technologies and the domain. The lead developer typically belongs to the virtual architectural team led by the system architect. There is no point in having a team with just junior developers or just senior developers. The idea is to transfer skills and knowledge from senior developers to junior developers. This also works the other way around. Junior developers can have knowledge of some of the latest technologies and practices that senior developers are missing. So overall, the best team is a team consisting of a good mix of both junior and senior developers.

Test Automation Developer

A test automation developer is responsible for developing different kinds of automated tests. Typically, test automation developers develop integration, E2E, and automated non-functional tests. A test automation developer must have good proficiency in at least one programming language, like Python, that is used to develop automated tests. Test automation developers must have a good command of BDD and some common testing frameworks, like Cucumber-JVM or Behave. Knowledge of some testing tools, like Apache JMeter, is appreciated. Test automation developers can also develop internal testing tools, like interface simulators and data generators. Test automation developers should form a virtual team to facilitate the development of E2E and automated non-functional tests.

DevOps Engineer

A DevOps engineer acts as an interface between the software development team and the software operations. A DevOps engineer usually creates CI/CD pipelines for microservices and crafts infrastructure and deployment-related code. DevOps engineers also define alerting rules and metrics visualization dashboards that can be used when monitoring the software in production. DevOps engineers help operations personnel to monitor software in production. They can help troubleshoot problems that the technical support organization cannot solve. DevOps engineer knows the environment (=infrastructure and platform) where the software is deployed, meaning that basic knowledge of at least one cloud provider (AWS/Azure/Google Cloud, etc.) and perhaps Kubernetes is required. DevOps engineers should form a virtual team to facilitate specifying DevOps-related practices and guidelines.

UI Designer

A UI designer is responsible for designing the final UIs based on higher-level UX/UI designs/wireframes. The UI designer will also conduct usability testing of the software.

DevSecOps

DevOps describes practices that integrate software development (Dev) and software operations (Ops). It aims to shorten the software development lifecycle through development parallelization and automation and provides continuous delivery of high-quality software. DevSecOps enhances DevOps by adding security aspects to the software lifecycle.

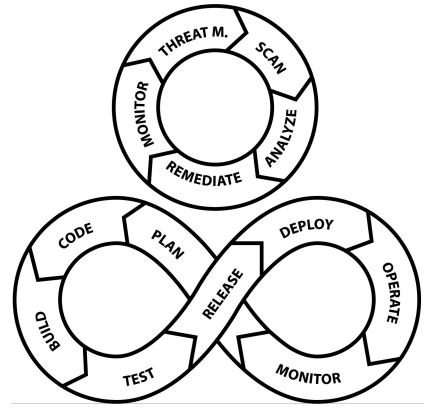

Fig 11.1 DevSecOps Diagram

A software development organization is responsible for planning, designing, and implementing software deliverables. Software operations deploy software to IT infrastructure and platforms. They monitor the deployed software to ensure it runs without problems. Software operations also provide feedback to the software development organization through bug reports and enhancement ideas.

SecOps Lifecycle

The SecOps lifecycle is divided into the following phases:

- Threat modeling
    - To find out what kind of security features and tests are needed
    - Implementation of threat countermeasures and mitigation. This aspect was covered in more detail in the earlier *security principles* chapter
- Scan
    - Static security analysis (also known as SAST = Static Application Security Testing)
    - Security testing (also known as DAST = Dynamic Application Security Testing)
    - Container vulnerability scanning
- Analyze
    - Analyze the results of the scanning phase, detect and remove false positives and prioritize corrections of vulnerabilities
- Remediate
    - Fix the found vulnerabilities according to prioritization
- Monitor
    - Define SecOps-related metrics and monitor them

DevOps Lifecycle

The DevOps lifecycle is divided into the following phases:

- Plan
- Code
- Build
- Test
- Release
- Deploy
- Operate
- Monitor

Subsequent sections describe each of the phases in more detail.

Plan

Plan is the first phase in the DevOps lifecycle. In this phase, software features are planned, and high-level architecture and UX are designed. This phase involves business (product management) and software development organizations.

Code

Code is the software implementation phase. It consists of software components' design and implementation, writing unit tests, integration tests, E2E tests, and other automated tests. This phase also includes all other coding needed to make the software deployable. Most of the work is done in this phase, so it should be streamlined as much as possible.

The key to shortening this phase is to parallelize everything to the maximum possible extent. In the *Plan* phase, the software was architecturally split into smaller pieces (microservices) that different teams could develop in parallel. Regarding developing a single microservice, there should also be as much parallelization as possible. This means that if a microservice can be split into multiple subdomains, the development of these subdomains can be done very much in parallel. If we think about the data exporter microservice, we identified several subdomains: input, decoding, transformations, encoding, and output. If you can parallelize the development of these five subdomains, you can significantly shorten the time needed to complete the implementation of the microservice.

To shorten this phase even more, a team should have a dedicated test automation developer who can start developing automated tests in an early phase parallel to the implementation.

Providing high-quality software relies on high-quality design, implementation with little technical debt, and comprehensive functional and non-functional testing. All of these aspects were already handled in the earlier chapters.

Build and Test

The *Build and Test* phase should be automated and run as *continuous integration* (CI) pipelines. Each software component in a software system should have its own CI pipeline. A CI pipeline is run by a CI tool like *Jenkins* or *Github Actions*. A CI pipeline is defined using declarative code stored in the software component's source code repository. Every time a commit is made to the main branch in the source code repository, it should trigger a CI pipeline run.

The CI pipeline for a software component should perform at least the following tasks:

- Checkout the latest source code from the source code repository
- Build the software
- Perform static code analysis. A tool like *SonarQube*/*SonarCloud* can be used
    - Perform static application security testing (SAST).
- Execute unit tests
- Execute integration tests
- Perform dynamic application security testing (DAST). A tool like *OWASP ZAP* can be used
- Verify 3rd party license compliance and provide a bill of materials (BOM). A tool like *Fossa* can be used

Release

In the *Release* phase, built and tested software is released automatically. After a software component's CI pipeline is successfully executed, the software component can be automatically released. This is called *continuous delivery* (CD). Continuous delivery is often combined with the CI pipeline to create a CI/CD pipeline for a software component. Continuous delivery means that the software component's artifacts are delivered to artifact repositories, like Artifactory, Docker Hub, or a Helm chart repository.

A CD pipeline should perform the following tasks:

- Perform static code analysis for the code that builds a container image (e.g., *Dockerfile*). A tool like *Hadolint* can be used for Dockerfiles.
- Build a container image for the software component
- Publish the container image to a container registry (e.g., Docker Hub, Artifactory, or a registry provided by your cloud provider)
- Perform a container image vulnerability scan
    - Remember to enable container vulnerability scanning at regular intervals in your container registry, also
- Perform static code analysis for deployment code. Tools like Helm's *lint* command, *Kubesec* and *Checkov* can be used
- Package and publish the deployment code (for example, package a Helm chart and publish it to a Helm chart repository)

Example Dockerfile

Below is an example *Dockerfile* for a microservice written in TypeScript for Node.js. The Dockerfile uses Docker's multi-stage feature. First (at the builder stage), it builds the source code, i.e., transpiles TypeScript source code files to JavaScript source code files. Then (at the intermediate stage), it creates an intermediate image that copies the built source code from the builder stage and

installs only the production dependencies. The last stage (final) copies files from the intermediate stage to a distroless Node.js base image. You should use a distroless base image to make the image size and the attack surface smaller. A distroless image does not contain any Linux distribution inside it.

Dockerfile

```
# syntax=docker/dockerfile:1

FROM node:18 as builder
WORKDIR /microservice
COPY package*.json ./
RUN  npm ci
COPY tsconfig*.json ./
COPY src ./src
RUN npm run build

FROM node:18 as intermediate
WORKDIR /microservice
COPY package*.json ./
RUN npm ci --only=production
COPY --from=builder /microservice/build ./build

FROM gcr.io/distroless/nodejs:18 as final
WORKDIR /microservice
USER nonroot:nonroot
COPY --from=intermediate --chown=nonroot:nonroot /microservice ./
CMD ["build/main"]
```

Example Kubernetes Deployment

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "microservice.fullname" . }}
  labels:
    {{- include "microservice.labels" . | nindent 4 }}
spec:
  {{- if ne .Values.nodeEnv "production" }}
  replicas: 1
  {{- end }}
  selector:
    matchLabels:
      {{- include "microservice.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      {{- with .Values.deployment.pod.annotations }}
      annotations:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      labels:
        {{- include "microservice.selectorLabels" . | nindent 8 }}
    spec:
      {{- with .Values.deployment.pod.imagePullSecrets }}
      imagePullSecrets:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      serviceAccountName: {{ include "microservice.serviceAccountName" . }}
```

```
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.imageRegistry }}/{{ .Values.imageRepository }}:
{{ .Values.imageTag }}"
          imagePullPolicy: {{ .Values.deployment.pod.container.imagePullPolicy }}
          securityContext:
            {{- toYaml .Values.deployment.pod.container.securityContext | nindent 12 }}
          {{- if .Values.httpServer.port }}
          ports:
            - name: http
              containerPort: {{ .Values.httpServer.port }}
              protocol: TCP
          {{- end }}
          env:
            - name: NODE_ENV
              value: {{ .Values.nodeEnv }}
            - name: ENCRYPTION_KEY
              valueFrom:
                secretKeyRef:
                  name: {{ include "microservice.fullname" . }}
                  key: encryptionKey
            - name: MICROSERVICE_NAME
              value: {{ include "microservice.fullname" . }}
            - name: MICROSERVICE_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: MICROSERVICE_INSTANCE_ID
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: NODE_NAME
              valueFrom:
                fieldRef:
                  fieldPath: spec.nodeName
            - name: MYSQL_HOST
              value: {{ .Values.database.mySql.host }}
            - name: MYSQL_PORT
              value: "{{ .Values.database.mySql.port }}"
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  name: {{ include "microservice.fullname" . }}
                  key: mySqlUser
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: {{ include "microservice.fullname" . }}
                  key: mySqlPassword
          livenessProbe:
            httpGet:
              path: /isMicroserviceAlive
              port: http
            failureThreshold: 3
            periodSeconds: 10
          readinessProbe:
            httpGet:
              path: /isMicroserviceReady
              port: http
            failureThreshold: 3
            periodSeconds: 5
          startupProbe:
            httpGet:
```

```
              path: /isMicroserviceStarted
              port: http
            failureThreshold:
{{ .Values.deployment.pod.container.startupProbe.failureThreshold }}
            periodSeconds: 10
          resources:
            {{- if eq .Values.nodeEnv "development" }}
            {{- toYaml .Values.deployment.pod.container.resources.development | nindent 12 }}
            {{- else if eq .Values.nodeEnv "integration"  }}
            {{- toYaml .Values.deployment.pod.container.resources.integration | nindent 12 }}
            {{- else }}
            {{- toYaml .Values.deployment.pod.container.resources.production | nindent 12 }}
            {{- end}}
      {{- with .Values.deployment.pod.nodeSelector }}
      nodeSelector:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.deployment.pod.affinity }}
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchLabels:
                  app.kubernetes.io/name: {{ include "microservice.name" . }}
              topologyKey: "kubernetes.io/hostname"
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.deployment.pod.tolerations }}
      tolerations:
        {{- toYaml . | nindent 8 }}
      {{- end }}
```

The values (indicated by `.Values.<something>`) in the above template come from a *values.yaml* file. Below is an example *values.yaml* file to be used with the above Helm chart template.

values.yaml

```
imageRegistry: docker.io
imageRepository: pksilen2/backk-example-microservice
imageTag:
nodeEnv: production
auth:
  # Authorization Server Issuer URL
  # For example
  # http://keycloak.platform.svc.cluster.local:8080/auth/realms/<my-realm>
  issuerUrl:

  # JWT path where for user's roles,
  # for example 'realm_access.roles'
  jwtRolesClaimPath:
secrets:
  encryptionKey:
database:
  mySql:
    # For example:
    # my-microservice-mysql.default.svc.cluster.local or
    # cloud database host
    host:
    port: 3306
    user:
    password: &mySqlPassword ""
```

```
mysql:
  auth:
    rootPassword: *mySqlPassword
deployment:
  pod:
    annotations: {}
    imagePullSecrets: []
    container:
      imagePullPolicy: Always
      securityContext:
        privileged: false
        capabilities:
          drop:
            - ALL
        readOnlyRootFilesystem: true
        runAsNonRoot: true
        runAsUser: 65532
        runAsGroup: 65532
        allowPrivilegeEscalation: false
      env:
      startupProbe:
        failureThreshold: 30
      resources:
        development:
          limits:
            cpu: '1'
            memory: 768Mi
          requests:
            cpu: '1'
            memory: 384Mi
        integration:
          limits:
            cpu: '1'
            memory: 768Mi
          requests:
            cpu: '1'
            memory: 384Mi
        production:
          limits:
            cpu: 1
            memory: 768Mi
          requests:
            cpu: 1
            memory: 384Mi
    nodeSelector: {}
    tolerations: []
    affinity: {}
```

Especially notice the `deployment.pod.container.securityContext` object in the above file. It is used to define the security context for a microservice container.

By default, the security context should be the following:

- Container should not be privileged
- All capabilities are dropped
- Container filesystem is read-only
- Only a non-root user is allowed to run inside the container

- Define the non-root user and group under which the container should run
- Disallow privilege escalation

You can remove things from the above list only if it is mandatory for a microservice. For example, if a microservice must write to the filesystem for some valid reason, then the filesystem should not be defined as read-only.

Example CI/CD Pipeline

Below is a GitHub Actions CI/CD workflow for a Node.js microservice. The declarative workflow is written in YAML. The workflow file should be located in the microservice's source code repository in the *.github/workflows* directory. Steps in the workflow are described in more detail after the example.

```
name: CI/CD workflow
on:
  workflow_dispatch: {}
  push:
    branches:
      - main
    tags-ignore:
      - '**'
jobs:
  build:
    runs-on: ubuntu-latest
    name: Build with Node version 18
    steps:
      - name: Checkout Git repo
        uses: actions/checkout@v2

      - name: Setup Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '18'
          cache: 'npm'

      - name: Install NPM dependencies
        run: npm ci

      - name: Lint source code
        run: npm run lint

      - name: Run unit tests with coverage
        run: npm run test:coverage

      - name: Setup integration testing environment
        run: docker-compose --env-file .env.ci up --build -d

      - name: Run integration tests
        run: scripts/run-integration-tests-in-ci.sh

      - name: OWASP ZAP API scan
        uses: zaproxy/action-api-scan@v0.1.0
        with:
          target: generated/openapi/openApiPublicSpec.yaml
          fail_action: true
```

```
          cmd_options: -I -z "-config replacer.full_list(0).description=auth1
                              -config replacer.full_list(0).enabled=true
                              -config replacer.full_list(0).matchtype=REQ_HEADER
                              -config replacer.full_list(0).matchstr=Authorization
                              -config replacer.full_list(0).regex=false
                              -config 'replacer.full_list(0).replacement=Bearer
ZXlK...aGJHZ='"

      - name: Tear down integration testing environment
        run: docker-compose --env-file .env.ci down -v

      - name: Static code analysis with SonarCloud scan
        uses: sonarsource/sonarcloud-github-action@v1.6
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}

      - name: 3rd party software license compliance analysis with FOSSA
        uses: fossas/fossa-action@v1
        with:
          api-key: ${{ secrets.FOSSA_API_KEY }}
          run-tests: false

      - name: Lint Dockerfile
        uses: hadolint/hadolint-action@v1.6.0

      - name: Log in to Docker registry
        uses: docker/login-action@v1
        with:
          registry: docker.io
          username: ${{ secrets.DOCKER_REGISTRY_USERNAME }}
          password: ${{ secrets.DOCKER_REGISTRY_PASSWORD }}

      - name: Extract latest Git tag
        uses: actions-ecosystem/action-get-latest-tag@v1
        id: extractLatestGitTag

      - name: Set up Docker Buildx
        id: setupBuildx
        uses: docker/setup-buildx-action@v1

      - name: Cache Docker layers
        uses: actions/cache@v2
        with:
          path: /tmp/.buildx-cache
          key: ${{ runner.os }}-buildx-${{ github.sha }}
          restore-keys: |
            ${{ runner.os }}-buildx-

      - name: Extract metadata for building and pushing Docker image
        id: dockerImageMetadata
        uses: docker/metadata-action@v3
        with:
          images: ${{ secrets.DOCKER_REGISTRY_USERNAME }}/example-microservice
          tags: |
            type=semver,pattern={{version}},value=$
{{ steps.extractLatestGitTag.outputs.value }}

      - name: Build and push Docker image
        id: dockerImageBuildAndPush
        uses: docker/build-push-action@v2
        with:
          context: .
```

```
          builder: ${{ steps.setupBuildx.outputs.name }}
          push: true
          cache-from: type=local,src=/tmp/.buildx-cache
          cache-to: type=local,dest=/tmp/.buildx-cache
          tags: ${{ steps.dockerImageMetadata.outputs.tags }}
          labels: ${{ steps.dockerImageMetadata.outputs.labels }}

      - name: Docker image vulnerability scan with Anchore
        id: anchoreScan
        uses: anchore/scan-action@v3
        with:
          image: ${{ secrets.DOCKER_REGISTRY_USERNAME }}/example-microservice:latest
          fail-build: false
          severity-cutoff: high

      - name: Upload Anchore scan SARIF report
        uses: github/codeql-action/upload-sarif@v1
        with:
          sarif_file: ${{ steps.anchoreScan.outputs.sarif }}

      - name: Install Helm
        uses: azure/setup-helm@v1
        with:
          version: v3.7.2

      - name: Extract microservice version from Git tag
        id: extractMicroserviceVersionFromGitTag
        run: |
          value="${{ steps.extractLatestGitTag.outputs.value }}"
          value=${value:1}
          echo "::set-output name=value::$value"

      - name: Update Helm chart versions in Chart.yaml
        run: |
          sed -i "s/^version:.*/version: $
{{ steps.extractMicroserviceVersionFromGitTag.outputs.value }}/g" helm/example-microservice/
Chart.yaml
          sed -i "s/^appVersion:.*/appVersion: $
{{ steps.extractMicroserviceVersionFromGitTag.outputs.value }}/g" helm/example-microservice/
Chart.yaml

      - name: Update Docker image tag in values.yaml
        run: |
          sed -i "s/^imageTag:.*/imageTag:
{{ steps.extractMicroserviceVersionFromGitTag.outputs.value }}@$
{{ steps.dockerImageBuildAndPush.outputs.digest }}/g" helm/example-microservice/values.yaml

      - name: Lint Helm chart
        run: helm lint -f helm/values/values-minikube.yaml helm/example-microservice

      - name: Static code analysis for Helm chart with Checkov
        uses: bridgecrewio/checkov-action@master
        with:
          directory: helm/example-microservice
          quiet: false
          framework: helm
          soft_fail: false

      - name: Upload Checkov SARIF report
        uses: github/codeql-action/upload-sarif@v1
        with:
          sarif_file: results.sarif
          category: checkov-iac-sca
```

```
    - name: Configure Git user
      run: |
        git config user.name "$GITHUB_ACTOR"
        git config user.email "$GITHUB_ACTOR@users.noreply.github.com"

    - name: Package and publish Helm chart
      uses: helm/chart-releaser-action@v1.2.1
      with:
        charts_dir: helm
      env:
        CR_TOKEN: "${{ secrets.GITHUB_TOKEN }}"
```

1. Checkout the microservice's Git repository
2. Setup Node.js 18
3. Install NPM dependencies
4. Lint source code using the `npm run lint` command, which uses ESLint
5. Execute unit tests and report coverage
6. Set up an integration testing environment using Docker's `docker-compose up` command. After executing the command the microservice is built, and all the dependencies in separate containers are started. These dependencies can include other microservices and, for example, a database and a message broker, like Apache Kafka
7. Execute integration tests. This script will first wait until all dependencies are up and ready. This waiting is done running a container using the *dokku/wait* (https://hub.docker.com/r/dokku/wait) image.
8. Perform DAST with OWASP ZAP API scan. For the scan, we define the location of the OpenAPI 3.0 specification file against which the scan will be made. We also give command options to set a valid Authorization header for the HTTP requests made by the scan
9. Tear down the integration testing environment
10. Perform static code analysis using SonarCloud
11. Check 3rd party software license compliance using FOSSA
12. Lint Dockerfile
13. Log in to Docker Hub
14. Extract the latest Git tag for further use
15. Setup Docker Buildx and cache Docker layers
16. Extract metadata, like the tag and labels for building and pushing a Docker image
17. Build and push a Docker image
18. Perform a Docker image vulnerability scan with Anchore
19. Upload the Anchore scan report to the GitHub repository
20. Install Helm
21. Extract the microservice version from the Git tag (remove the 'v' letter before the version number)
22. Replace Helm chart versions in the Helm chart's *Chart.yaml* file using the `sed` command

23. Update the Docker image tag in the *values.yaml* file
24. Lint the Helm chart and perform static code analysis for it
25. Upload the static code analysis report to the GitHub repository and perform git user configuration for the next step
26. Package the Helm chart and publish it to GitHub Pages

Some of the above steps are parallelizable, but a GitHub Actions workflow does not currently support parallel steps in a job. In Jenkins, you can easily parallelize stages using a *parallel* block.

You could also execute the unit tests and linting when building a Docker image by adding the following steps to the *builder* stage in the *Dockerfile*:

```
RUN npm run lint
RUN npm run test:coverage
```

The problem with the above solution is that you don't get a clear indication of what failed in a build. You must examine the output of the Docker build command to see if linting or unit tests failed. Also, you cannot use the SonarCloud GitHub Action anymore. You must implement SonarCloud reporting in the *builder* stage of the Dockerfile (after completing the unit testing to report the unit test coverage to SonarCloud).

Deploy

In the *Deploy* phase, released software is deployed automatically. After a successful CI/CD pipeline run, a software component can be automatically deployed. This is called *continuous deployment* (CD). Notice that both *continuous delivery* and *continuous deployment* are abbreviated as CD. This can cause unfortunate misunderstandings. Continuous delivery is about releasing software automatically, and continuous deployment is about automatically deploying released software to one or more environments. These environments include, for example, a CI/CD environment, staging environment(s) and finally, production environment(s). There are different ways to automate software deployment. One modern and popular way is to use GitOps, which uses a Git repository or repositories to define automatic deployments to different environments using a declarative approach. GitOps can be configured to update an environment automatically when new software is released. This is typically done for the CI/CD environment, which should always be kept up-to-date and contain the latest software component versions.

GitOps can also be configured to deploy automatically and regularly to a staging environment. A staging environment replicates a production environment. It is an environment where end-to-end functional and non-functional tests are executed before the software is deployed to production. You can use multiple staging environments to speed up the continuous deployment to production. It is vital that all needed testing is completed before deploying to production. Testing can take a couple of days to validate the stability of the software. If testing in a staging environment requires three

days and you set up three staging environments, you can deploy to production every day. On the other hand, if testing in a staging environment takes one week and you have only one staging environment, you can deploy to production only once a week (Assuming here that all tests execute successfully) Deployment to a production environment can also be automated. Or it can be triggered manually after successfully completing all testing in a staging environment.

Operate

Operate is the phase when the software runs in production. In this phase, it needs to be secured that software updates (like security patches) are timely deployed. Also, the production environment's infrastructure and platform should be kept up-to-date and secure.

Monitor

Monitor is the phase when a deployed software system is monitored to detect any possible problems. Monitoring should be automated as much as possible. It can be automated by defining rules for alerts triggered when the software system operation requires human intervention. These alerts are typically based on various metrics collected from the microservices, infrastructure, and platform. *Prometheus* is a popular system for collecting metrics, visualizing them, and triggering alerts.

The basic monitoring workflow follows the below path:

1. Monitor alerts
2. If an alert is triggered, investigate metrics in relevant dashboards
3. Check logs for errors in relevant services
4. Distributed tracing can help to visualize if and how requests between different microservices are failing

The following needs to be done to make monitoring possible and easy:

- Each service must log to the standard output
    - If your microservice is using a 3rd party library that logs to the standard output, choose a library that allows you to configure the logging format or request the log format configurability as an enhancement to the library
    - Choose a standardized log format and use it in all microservices, e.g., *Syslog* format or OpenTelemetry Logs Data Model (https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/logs/data-model.md)
    - Collect logs from each microservice to a centralized location, like an ElasticSearch database

- Integrate microservices to a distributed tracing tool, like Jaeger
    - A distributed tracing tool collects information about network requests microservices make
- Define what metrics are needed to be collected from each microservice
    - Collect metrics that are needed to calculate the most important *service level indicators* (SLIs):
        - Availability
        - Error rate
        - Latency
        - Throughput
    - Instrument your microservice with the necessary code to collect the metrics. This can be done using a metrics collection library, like Prometheus
- Define how to visualize metrics
    - Create a main dashboard for each microservice to present the SLIs. You must also present *service level objectives* (SLOs).
    - When all SLOs are met, the dashboard should show SLI values in green. If an SLO is not met, the corresponding SLI value should be shown in red. You can also use yellow and orange colors to indicate that an SLO is still met, but the SLI value is no more optimal.
    - Create troubleshooting dashboards, if needed.
    - Use a visualization tool that integrates with the metrics collection tool, like Grafana with Prometheus.
    - You can usually deploy metric dashboards as part of the microservice deployment. If you are using Kubernetes, Prometheus and Grafana, you can create Grafana dashboards as custom resources (CRs) when using the Grafana Operator (https://github.com/grafana-operator/grafana-operator)
- Define alerting rules
    - First, define the service level objectives (SLOs) and base the alerting rules on them
    - An example of an SLO: "service error rate must be less than x percent"
    - If an SLO cannot be met, an alert should be triggered
    - If you are using Kubernetes and Prometheus, you can define alerts using the Prometheus Operator and PrometheusRule CRs.

Software operations staff connects back to the software development side of the DevOps lifecycle in the following ways:

- Ask for technical support
- File a bug report
- File an improvement idea

The first one will result in a solved case or bug report. The latter two will reach the *Plan* phase of

the DevOps lifecycle. Bug reports usually enter the *Code* phase immediately, depending on the fault severity.

Logging

Implement logging in software components using the following logging severities:

- (CRITICAL/FATAL)
- ERROR
- WARNING
- INFO
- DEBUG
- TRACE

I don't usually use the CRITICAL/FATAL severity at all. It is better to report all errors with the ERROR severity because then it is easy to query logs for errors using a single keyword, for example:

```
kubectl logs <pod-name> | grep ERROR
```

You can add information to the log message itself about the criticality/fatality of an error. When you log an error for which there is a solution available, you should inform the user about the solution in the log message, e.g., provide a link to a troubleshooting guide or give an error code that can be used to search the troubleshooting guide.

Do not log too much information using the INFO severity because the logs might be difficult to read when there is too much noise. Consider carefully what should be logged with the INFO severity and what can be logged with the DEBUG severity instead. The default logging level of a microservice should be WARNING or INFO.

Use the TRACE severity to log only tracing information, e.g., detailed information related to processing a single request, event, or message.

OpenTelemetry Log Data Model

This section describes the essence of the OpenTelemetry log data model version 1.12.0 (Please check https://github.com/open-telemetry/opentelemetry-specification for possible updates).

A log entry is a JSON object containing the following properties:

Property Name	Description
Timestamp	Time when the event occurred. Nanoseconds since Unix epoch
TraceId	Request trace id
SpanId	Request span id
SeverityText	The severity text (also known as log level)
SeverityNumber	Numerical value of the severity
Body	The body of the log entry. You can include ISO 8601 timestamp and the severity/log level before the actual log message
Resource	Describes the source of the log entry
Attributes	Additional information about the log event. This is a JSON object where custom attributes can be given

Below is an example log entry according to the OpenTelemetry log data model.

```
{
  "Timestamp": "1586960586000000000",
  "TraceId": "f4dbb3edd765f620",
  "SpanId": "43222c2d51a7abe3",
  "SeverityText": "ERROR",
  "SeverityNumber": 9,
  "Body": "20200415T072306-0700 ERROR Error message comes here",
  "Resource": {
    "service.namespace": "default",
    "service.name": "my-microservice",
    "service.version": "1.1.1",
    "service.instance.id": "my-microservice-34fggd-56faae"
  },
  "Attributes": {
    "http.status_code": 500,
    "http.url": "http://example.com",
    "myCustomAttributeKey": "myCustomAttributeValue"
  }
}
```

The above JSON-format log entries might be hard to read as plain text on the console, for example, when viewing a pod's logs with the `kubectl logs` command in a Kubernetes cluster. You can create a small script that extracts only the `Body` property value from each log entry.

PrometheusRule Example

PrometheusRule custom resources (CRs) can be used to define rules for triggering alerts. In the below example, an *example-microservice-high-request-latency* alert will be triggered with a major severity when the median request latency in seconds is greater than one (`request_latencies_in_seconds{quantile="0.5"} > 1`).

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: example-microservice-rules
spec:
  groups:
  - name: example-microservice-rules
    rules:
    - alert: example-microservice-high-request-latency
      expr: request_latencies_in_seconds{quantile="0.5"} > 1
      for: 10m
      labels:
        application: example-microservice
        severity: major
        class: latency
      annotations:
        summary: "High request latency on {{ $labels.instance }}"
        description: "{{ $labels.instance }} has a median request latency above 1s (current
value: {{ $value }}s)"
```

Appendix A

This appendix presents the implementation of an object validation library in TypeScript (version 4.7.4 used). This library uses the *validated-types* NPM library as a basis.

This object validation library will be able to validate a JSON object whose schema is given in the below format. The schema object lists the properties of the object to be validated. The value of each property is the schema for that property in the object to be validated. In this example, we implement support for validating integers, strings, and nested objects. Validation of other types (e.g., floats and arrays) could also be added.

The schema of an integer property is defined as follows:

```
['int' | 'int?',  'min-value,max-value']
```

The schema of a string property is defined using one of the three syntaxes as shown below:

```
['string' | 'string?',  'min-length,max-length,validator']

['string' | 'string?',  ['min-length,max-length,validator',
                         'validator']]


['string' | 'string?',  ['min-length,max-length,validator',
                         'validator',
                         'validator']]
```

Below is an example schema object:

```
const schema = {
  inputUrl: ['string', ['1,1024,url',
                        'startsWith,https',
                        'endsWith,com']] as const,

  outputPort: ['int?', '1,65535'] as const,
  mongoDb: {
    user: ['string?', '1,512'] as const
```

```
  },
  transformations: [
    {
      outputFieldName: ['string', '1,512'] as const,
      inputFieldName: ['string', '1,512'] as const
    }
  ] as const
};

const defaultValues = {
  outputPort: 8080
};
```

Explanations for individual property schemas from above:

- `inputUrl`
    - Mandatory string property
    - Length must be between 1-1024 characters
    - Value must be an URL and start with 'https' and end with 'com'
- `outputPort`
    - Optional integer property
    - Value must be between 1-65535
- `mongoDb.user`
    - Optional string property
    - Length must be between 1-512 characters
- `transformations[*].outputFieldName`
    - Mandatory string property
    - Length must be between 1-512 characters
- `transformations[*].inputFieldName`
    - Mandatory string property
    - Length must be between 1-512 characters

Below are the definitions of TypeScript types needed for the library:

ValidatedObject.ts

```
import { VInt, VString } from 'validated-types';

type Type = string;
type ValidationSpec = string | [string, string] | [string, string, string];

type ValidationClass<T extends Type, V extends ValidationSpec> = T extends `${infer
NullableType}?`
  ? NullableType extends 'string'
    ? VString<V> | null
    : NullableType extends 'int'
    ? V extends string
      ? VInt<V> | null
      : never
```

```
    : never
  : T extends `${infer T}`
  ? T extends 'string'
    ? VString<V>
    : T extends 'int'
    ? V extends string
      ? VInt<V>
      : never
    : never
  : never;

type PropertySchema = [Type, ValidationSpec];
export type ObjectSchema = { [propertyName: string]: PropertySchema | ObjectSchema |
[ObjectSchema] };

export type ValidatedObject<S extends ObjectSchema> = {
  [P in keyof S]: S[P] extends [ObjectSchema]
    ? Array<ValidatedObject<S[P][0]>>
    : S[P] extends ObjectSchema
    ? ValidatedObject<S[P]>
    : S[P] extends PropertySchema
    ? ValidationClass<S[P][0], S[P][1]>
    : never;
};
```

The below `tryCreateValidateObject` function tries to create a validated version of a given object:

tryCreateValidateObject.ts

```
import { VInt, VString } from 'validated-types';
import { DeepWritable, Writable } from 'ts-essentials';
import { ObjectSchema, ValidatedObject } from './ValidatedObject';

function tryCreateValidatedObject<S extends ObjectSchema>(
  object: Record<string, any>,
  objectSchema: OS,
  defaultValuesObject?: Record<string, any>
): ValidatedObject<S> {
  const validatedObject = {};

  Object.entries(objectSchema as Writable<S>).forEach(([propertyName,
                                                        objectOrPropertySchema]) => {
    const isPropertySchema = Array.isArray(objectOrPropertySchema) &&
                             typeof objectOrPropertySchema[0] === 'string';

    if (isPropertySchema) {
      const [propertyType, propertySchema] = objectOrPropertySchema;
      if (propertyType.startsWith('string')) {
        (validatedObject as any)[propertyName] = propertyType.endsWith('?')
          ? VString.create(propertySchema, object[propertyName] ??
              defaultValuesObject?.[propertyName])
          : VString.tryCreate(propertySchema, object[propertyName]);
      } else if (propertyType.startsWith('int') && !Array.isArray(propertySchema)) {
        (validatedObject as any)[propertyName] = propertyType.endsWith('?')
          ? VInt.create<typeof propertySchema>(
              propertySchema,
              object[propertyName] ?? defaultValuesObject?.[propertyName]
            )
          : VInt.tryCreate<typeof propertySchema>(propertySchema, object[propertyName]);
      }
```

```
    } else {
      if (Array.isArray(objectOrPropertySchema)) {
        (object[propertyName] ?? []).forEach((subObject: any) => {
          (validatedObject as any)[propertyName] = [
            ...((validatedObject as any)[propertyName] ?? []),
            tryCreateValidatedObject(subObject ?? {},
                                     objectOrPropertySchema[0],
                                     defaultValuesObject?.[propertyName])
          ];
        });
      } else {
        (validatedObject as any)[propertyName] = tryCreateValidatedObject(
          object[propertyName] ?? {},
          objectOrPropertySchema,
          defaultValuesObject?.[propertyName]
        );
      }
    }
  });

  return validatedObject as ValidatedObject<ObjSchema>;
}
```

Let's have the following unvalidated configuration object and create a validated version of it:

configuration.ts

```
const unvalidatedConfiguration = {
  inputUrl: 'https://www.google.com',
  mongoDb: {
    user: 'root'
  },
  transformations: [
    {
      outputFieldName: 'outputField',
      inputFieldName: 'inputField'
    }
  ]
};

// This will contain the validated configuration object with strong typing
let configuration: ValidatedObject<DeepWritable<typeof configurationSchema>>;

try {
  configuration = tryCreateValidatedObject(
    unvalidatedConfiguration,
    configurationSchema as DeepWritable<typeof configurationSchema>,
    configurationDefaultValues
  );

  console.log(validatedConfiguration.inputUrl.value);
  console.log(validatedConfiguration.outputPort?.value);
  console.log(validatedConfiguration.mongoDb.user?.value);
  console.log(validatedConfiguration.transformations[0].inputFieldName.value);
  console.log(validatedConfiguration.transformations[0].outputFieldName.value);
} catch (error) {
  // Handle error...
}

// We export the validated configuration object with strong typing
// to be used in other parts of the application
export default configuration;
```

Below is an example schema for a Node.js `process.env` object:

environmentSchema.ts

```
export const environmentSchema = {
  NODE_ENV: ['string', '1,32,isOneOf,["development","integration","production"]'],
  LOGGING_LEVEL: ['string?', '1,32,isOneOf,["DEBUG","INFO","WARN","ERROR"]'],
  MONGODB_PORT: ['string?', '1,5,numericRange,1-65535'],
  MONGODB_USER: [string?, '1,512']
}

export const environmentDefaultValues = {
  LOGGING_LEVEL: 'INFO',
  MONGODB_PORT: '27017'
}
```

We can create a validated environment from the `process.env`:

```
import { environmentDefaultValue, environmentSchema } from './environmentSchema';

let environment: ValidatedObject<DeepWritable<typeof environmentSchema>>;

try {
  environment = tryCreateValidatedObject(
    process.env,
    environmentSchema as DeepWritable<typeof environmentSchema>,
    environmentDefaultValues
  );
} catch (error) {
  // Handle error...
}

export default environment;
```