Jakarta EE for Java Developers

Build Cloud-Native and Enterprise Applications Using a High-Performance Enterprise Java Platform

RHUAN ROCHA

bpb

Jakarta EE for Java Developers

Build Cloud-Native and Enterprise Applications
Using a High-Performance Enterprise Java Platform

Rhuan Rocha

www.bpbonline.com

FIRST EDITION 2022

ISBN: 978-93-55510-082

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Dedicated to

Ivonete Rocha

My mother, who has believed in my dreams.

About the Author

Rhuan Rocha is an experiment Java engineer that has created many projects using the Java EE/Jakarta EE specifications. He has worked for a big Jakarta EE vendor called Red Hat with the Wildfly/JBoss EAP. Throughout your career, he has developed many Enterprise Java applications for private companies and governments in Brazil, using Java EE/Jakarta EE and technologies of its ecosystem.

Now, Rhuan Rocha is Principal Software Engineer at DigiBee developing its integration platform, a Jakarta EE specialist, and Co-founder of Cloud Conference Day. He is an Open Source contributor and contributed to JNoSQL, RestEasy, TomEE, Quarkus, and others.

About the Reviewer

Bauke Scholtz is an Oracle Java Champion, a member of the Jakarta Faces Expert Group and the main creator of the Jakarta Faces helper library OmniFaces. He is on the internet more commonly known as BalusC who is among the top contributors on Stack Overflow. He is a web application specialist and consults or has consulted for these Virtua.tech, Mercury1.co.uk, MyTutor.co.uk, LinkPizza.com, ZEEF.com, ITCA.com, RDC.nl and more clients from fintech, affiliate marketing, social media and more as part of his more than 20 years of experience.

Acknowledgement

Write a book is a hard task that starts long before typing the first word of the chapter. Accumulate knowledges and experiences is a big journey that requires many sacrifices. I really thank my family, closer friends, and teachers that have helped me in this journey. I want to make a special thank you to Bruna Grellt and João Purificação, that are big friends that have helped me in many things.

I would like to thank communities like SouJava and others that have helped me a lot to improve my knowledge, and a big thanks to Karina Varela that helped me a lot in my career.

Finally, I would like to thank BPB Publications for giving me this opportunity to write my first book for them.

Preface

The Java ecosystem has had many changes in its tools and approaches, generated by new challenges that came from Cloud Computing. Cloud Computing did force the developer to rethink its processes to developer application, its architectures, and approaches, and came to a new concept called Cloud Native Application. With this, the Java ecosystem needs the be rethinking as well, and the communities and companies started a big change in the Java ecosystem. One of these updates was the rebrand of Java EE to Jakarta EE and its new process to evolve this specification over the Eclipse Foundation.

In this book, we explain the rebrand of the Java EE to Jakarta EE, and how the new evolve process works. It is very important to understanding the Jakarta EE moment and to know what is expected by this specification. Furthermore, we talk about the changes applied between the Java EE 8 over Oracle and the Jakarta EE 9 over Eclipse Foundation.

This book has a practical approach and you will create one project per chapter. It enables you to read just the chapter you want to read because the chapters do not have a dependence between them. Each chapter is focused on one specification. Thus, this book can help you in many kinds of projects using Spring, Quarkus, Micronault, or others that use some Jakarta EE specification.

Chapter 1 introduces the Jakarta EE specification and its new process to evolve the specification by the Eclipse Foundation. At the end of this chapter, you will know about the environment and what is needed to start a project.

Chapter 2 shows you Jakarta Servlet using a practical way. Throughout this chapter you will learn how can we create a Servlet, how can we filtering requests and responses, what are Forward and RequestDispatcher and how can we use them, how can we work with the synchronous process, how can we work with Nonblocking I/O and how can we use Server Push.

Chapter 3 explain what is Jakarta Context and Dependency Injection (CDI) in a practical way. In this chapter you will learn how we can use CDI to injection bean, how the CDI life cycle works, how we can use the producer method and producer field to product object's instances, how we can use dispose method, how can we create a CDI Interceptor and how can we create a CDI Decorator.

Chapter 4 explains what is Jakarta RESTful Web Service using a practical way. In this chapter, you will learn how we can create a RESTful resource class, how we can extract request parameters, how we can use the Bean Validation with Jakarta RESTful Web Service, how we can create a RESTful Client, how we can make an asynchronous invocation in the client and how we can use a Server-Sent Event (SSE).

Chapter 5 explains what is the Jakarta Enterprise Bean covering the principal concepts and practicals. Besides, we'll explain how to

implement stateless bean, statefull bean, singleton bean and how it one manages transaction. At the end of the chapter, we will show you how we can create schedules of tasks and what is a statefull clustered.

Chapter 6 explains what is the Jakarta Persistence using a practical way. In this chapter, you will learn how can we make operations in a relational database, how we can mapping a table to entity, how we can use JPQL to make queries, how we can use EntityManager, how we can make a control of concurrent access to entity data and how we can make fetch plans using entity Graphs.

Chapter 7 explains what is the Jakarta Messaging and what are its aims. Besides that, throughout this chapter, we'll show you how we can create a Queue, how we can create a Topic, how we can create a producer and consumer, and finally, we will show how we can implement a Message-Driven Bean.

Chapter 8 explains what is Jakarta Security, how we can secure a web application, how we can implementing a user's authentication programmatically, how we can securing an enterprise bean, how we can implementing a custom identity store, and how we can implementing a custom database identity store.

Chapter 9 explains Jakarta Bean Validation using a practical way. In this chapter, you will learn how can we validate an entity using bean validation, how we can use it in a RESTful resource to validate data that came from requests. Besides that, we'll explain how we can validate constructors and methods and how can we

create a custom constraint. In the final, we will explain how we can customize the validator messages.

Downloading the code bundle and coloured images:

Please follow the link to download the ***Code Bundle*** and the ***Coloured Images*** of the book:

https://rebrand.ly/178bb3

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

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

At 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 BPB books and eBooks.

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit www.bpbonline.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.

The code bundle for the book is also hosted on GitHub at 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 Check them out!

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 business@bpbonline.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

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 BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

CHAPTER 1

Introduction to Jakarta EE

Introduction

The Java ecosystem has been used in many companies for many years and has provided several kinds of solutions to any environment. One of these environments is the enterprise environment, which has its particular necessities and challenges to be overcome. Thus, the Java ecosystem, by Sun — after being acquired by Oracle — promoted a project called J2EE, which is a project to provide well-known solutions to the common enterprise problems. Over time, its name changed to Java EE, and now it is Jakarta EE, which is maintained by the *Eclipse Foundation* now. This chapter will show you what the Jakarta EE is and what are its goals. Throughout this chapter, we'll walk through each main point of this project, analyzing the *why* for some changes and *what* can we expect from Jakarta EE. Besides that, we'll show you what a server application is, what its importance to the Jakarta EE ecosystem is, and what we need to work with Jakarta EE.

Structure

In this chapter, we will discuss the following topics:

Understanding what Jakarta EE is

From Java EE to Jakarta EE

Understanding the Jakarta EE tiers

Jakarta EE profiles

Server applications

Requirements for working with Jakarta EE

Objectives

The main objectives of the chapter are to enable the readers to understand what Jakarta EE is, what its goals are, and what is needed to work with it.

Understanding what Jakarta EE is

Basically, Jakarta EE is an umbrella project with a set of technologies for the enterprise Java applications, with an aim to provide the technologies and APIs to facilitate the development of the enterprise Java applications to the cloud computing world. This project came from the Java EE project, which was migrated from *Oracle* to *Eclipse* and is now evolving under the Jakarta EE brand. But to really understand what Jakarta EE is, we should take a look at their antecessors, Java EE and J2EE, and the current enterprise challenges. So, let's go!

J2EE and Java EE

In 1999, the *Sun Microsystem* was releasing the J2EE starting in its 1.2 version and taking the first step to put Java in the web world. The main goal of J2EE was to create an umbrella project based on the specifications, where these specifications describe the details of how some sets of technologies to the enterprise systems would work. With this, many vendors can create an implementation of these specifications and the developer can develop the enterprise Java application decoupled of the vendor and choose any vendor that follows the J2EE specifications and change the vendor as well. Thus, the J2EE came as a good solution to create the Java applications for the enterprise world, providing a set of robust technologies and more security to the business — now companies can create robust solutions without creating a dependency on any vendor.

J2EE was focused on the enterprise world and it has four elements in its architecture, these are described in the order of importance as follows:

J2EE It is a standard platform that follows the J2EE specifications and is the host to J2EE applications. It is known as an application server.

J2EE Compatibility Test A set of tests to verify if an implementation of J2EE follows the J2EE specifications.

J2EE Reference Each specification should have a reference implementation to demonstrate how the specification works.

Sun Blueprints Design Guidelines for It describes the standard programming model for the development of the multi-tier applications.

The first J2EE version came with these specifications – **Enterprise JavaBeans** Java Servlet, **Java Naming and Directory Interface JavaServer Pages Java Transaction Java Database Connectivity** and JavaMail. All these specifications exist and is part of Jakarta EE today, but with many improvements.

The J2EE had many evolutions in its versions J2EE 1.3 and J2EE 1.4, and then in 2006, its name was changed to Java EE starting with Java EE 5, after evolving to Java EE 6. Java EE 6 was the last version of Java EE under Sun Microsystem, because in 2010, *Oracle* bought *Sun Microsystem* and got the Java EE's rights. Java EE 6 was very important because it came after many evolutions to EJB, making its use easier, and it came to the **Context and Dependency Injection** that is the mechanism to dependencies control of Java EE. Java EE under the Oracle control had two versions released – Java EE 7 released in 2013 and Java EE 8 released in 2017.

As you can see, J2EE/Java EE is 20 years old and the enterprise world has evolved a lot in these years, and new challenges have emerged at the enterprise environment.

Cloud computing

The complexity of the enterprise world has grown very fast and new needs emerged with this. As the main necessities, we can quote the fault tolerance and elasticity of the business. In other words, the business needs to be able to respond effectively to the increasing demand, and it needs to do that with as smaller risks as possible. Today, the business innovation basis is the technology, and the technological world has been worked on to provide a set of models, approaches, and techniques to supply these needs. Cloud computing came as one of these solutions.

Cloud Computing is a set of techniques and approaches to serving a computational resource as a service with a better and easier way for the end-user. With cloud computing, we can have only a computational resource — such as memory, disk, CPU, node amount — that we really need and can scale up and down when needed. With this, the business can save money and generate a faster and more effective response to the increasing demand. Using cloud computing, we can have many types of services like **Platform as a Service Infrastructure as a Service Software as a Service Function as a Service** and others.

Cloud computing provides many benefits to the business, which are as follows:

Pay per The end-user pays only for the resources used. It can save a lot of money for the business.

The environment can recover after a failure.

Self-service The computational resource can be scaled up according to the demands in an easier and faster way, using an interface, eliminating the traditional need for the IT administrators to provision and manage to compute resources.

The computational resource can be scaled up or down at any time.

Fault As the environment can be run in many local areas, across many global regions, this environment has more fault tolerance.

Clearly, cloud computing comes with many benefits to the business world, but to take advantage of this, the application needs to be prepared to work in the cloud computing approach. But what does it mean?

The old approach used for a long time in the technological world has used a monolith architecture — all business domains or many business domains inside the same application — to develop the application. The result is a big application that has many responsibilities. *Figure 1.1* illustrates this as follows:

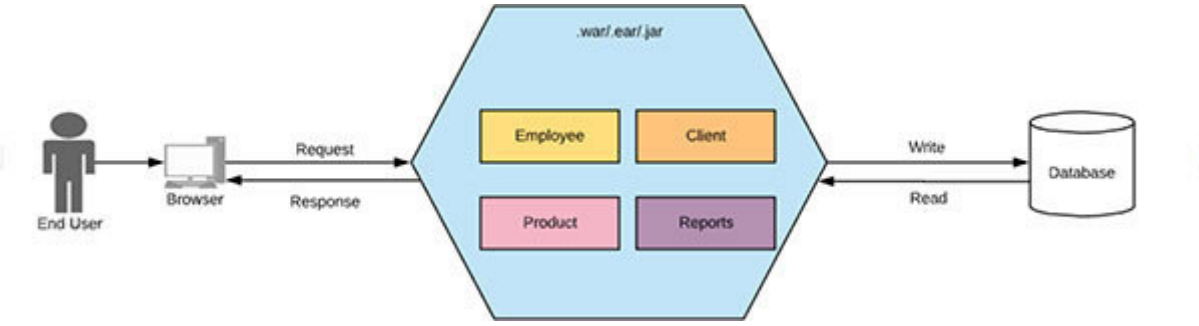

Figure 1.1: *Monoliths*

The monolith architecture has worked very well and works very well today, but outside a cloud computing environment. The monolith is not adequate to take advantage of cloud computing, because if you need to scale up this application, you'll scale all monolith, that is, you will scale all business domains inside monolith, and the benefit of *pay only for the resources used* will be lost. The resolution of this problem was rethinking how to architect the applications.

Microservice architecture

Rethinking about the architecture, the software engineers and architects noticed that to take advantage of cloud computing, he will have to break the monolith into the smallest parts with fine granularity to permit scaling only the part that really needs to be scaled. But how can it be broken? What are the metrics to define the boundaries of each part?

The microservice architecture breaks the monoliths into many small parts called microservice, which generally are broken according to the business domain, that is, each microservice has the responsibility to care about one business domain. Thus, the business domain is used to define the boundaries of the microservice. The use of the business domain as a base to define the microservices boundary is the approach widely used, but you can use another according to your scenario. But keep in mind that you should decompose to improve the ability to scale. *Figure 1.2* illustrates this architecture as follows:

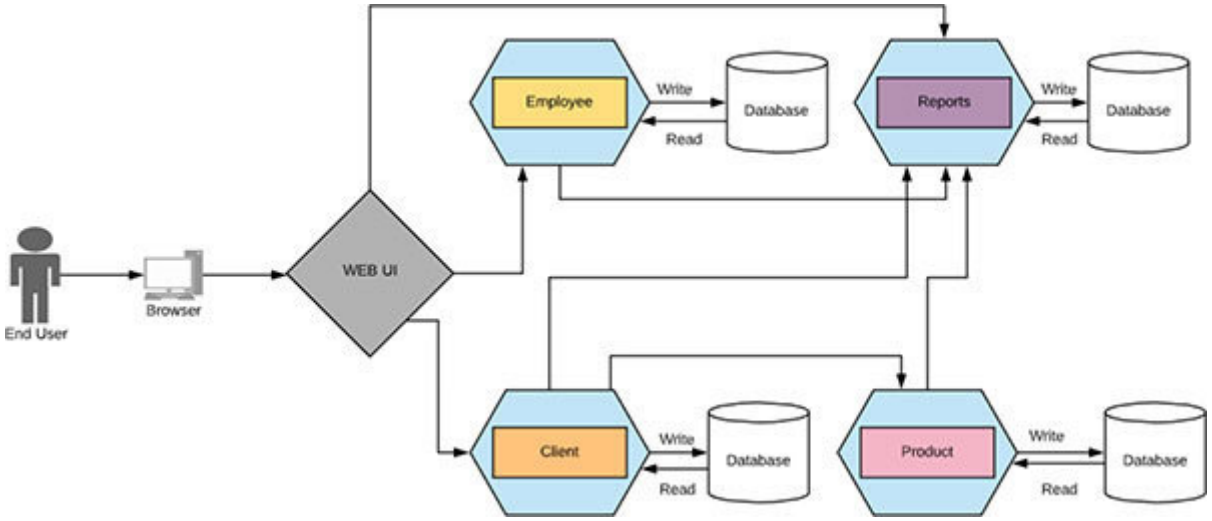

Figure 1.2: *Microservice Architecture*

In a microservice architecture, each microservice has low-coupling with each other, and they are independent and deployable. In other words, microservices are a small application that runs without the need for another microservice to do that. Those microservices integrate with each other when needed, to provide a response to the end-user. It is very good, but now the software engineers and architects have other problems. How will those microservices be managed and how will those applications be integrated with each other without becoming chaos? How to discover a new microservice when a new microservice service is born? How to load-balance those microservices and how will those microservices be scaled up and down without a side effect?

Twelve-factors

The twelve-factors is a methodology that defines a set of steps (twelve steps) to building the software-as-a-service apps. Basically, when we create the microservices apps and run those in a cloud environment, we are creating a software-as-a-service because our software is exposed as a service to the end-user. An application totally compliant with the cloud is called a cloud-native application.

The twelve-factors is a guide to develop the applications capable to take advantage of cloud computing. In other words, when we follow all the steps of the twelve-factors guide, our applications or microservices are a cloud-native application and is totally compliant with the cloud computing approach. The following are the twelve factors:

One codebase is tracked in the revision control, while many deploy.

Explicitly declare and isolate dependencies.

Store config in the environment.

Backing Treat the backing services as attached resources.

Build, release, Strictly separate the build and run stages.

Execute the app as one or more stateless processes.

Export the services via port binding.

Scale-out via the process model.

Maximize robustness with a fast startup and graceful shutdown.

Dev/prod Keep development, staging, and production as similar as possible.

Treat logs as event streams.

Admin Run admin/management tasks as one-off processes.

In this chapter's stage, you can note that we had many changes about how to architect the software. The applications have needed to make communication with each other and improving their management. But, coming back to Java EE, how does Java EE behave in these new scenarios?

Java EE had many gaps with those changes, because although Java EE has many strong specs, it does not have the specs to solve the problems about cloud computing and microservices. So, Java EE needed change.

Jakarta EE Goals

Java EE 8 was migrated to *Eclipse Foundation* and is now Jakarta EE with new goals and a new process to evolve.

Jakarta EE has a goal to accelerate the business application development for cloud computing, promoting many solutions based on the specifications worked by many vendors. Basically, Jakarta EE is a set of specifications with its **Technology Compatibility Kit** to test if an implementation is compliant with the specification. With this, all the vendors that create an implementation to Jakarta EE will only be granted as compliance to Jakarta EE if it is approved on test with TCK.

The first changed applied to Jakarta EE was the process of evolving. Java EE works with the **Java Community Process** to improve its features, but JCP is not compatible to attend to the changes promoted by cloud computing, because the changes in cloud computing happen fast and JCP's speed cannot keep up. Thus, Jakarta EE works with another process called **Jakarta EE Specification Process**

Jakarta EE Specification Process

Jakarta EE 8 started Jakarta EE in a new process called **Jakarta EE Specification Process** The JESP is the process used by the Jakarta EE Working Group — responsible for managing the changes and improvements on Jakarta EE — to evolve Jakarta EE. This process is based on **Eclipse Foundation Specification Process** with some changes. The following are those changes, described at

Any modification to or revision of this Jakarta EE Specification Process, including the adoption of a new version of the EFSP, must be approved by a super-majority of the specification committee, including a super-majority of the strategic members of the Jakarta EE Working Group, in addition to any other ballot requirements set forth in the EFSP.

All specification committee approval ballot periods will have the minimum duration, outlined as follows (notwithstanding the exception process defined by the EFSP, these periods may not be shortened):

Creation Review: 7 calendar days

Plan Review: 7 calendar days

Progress Review: 14 calendar days

Release Review: 14 calendar days

Service Release Review: 14 calendar days

JESP Update: 7 calendar days

A ballot will be declared invalid and concluded immediately in the event that the specification team withdraws from the corresponding review.

Specification projects must engage in at least one progress or release review per year while under active development.

JESP's goal is being a process as lightweight as possible working as an open-source process based on the code-first concept — the evolution is based on experimentation and the experiences gained with experimentation. This process drives the evolution of Jakarta EE, making Jakarta EE able to respond more quickly to the changes in the enterprise world.

Jakarta NoSQL

When we are writing a new application, we'll probably need to write a persistence module, because almost all applications integrate into the databases to persist the data. So, integrating to the database is a common problem in the enterprise application and Java EE created the **Java Persistence API** that is a Java EE solution to this common problem about database integration. In the past, almost all applications have used a relational database, so JPA was built with the relational databases in mind.

Now, in Jakarta EE, the Java persistence API became Jakarta persistence, but as we said, this specification was built with relational databases in mind. But, when the cloud computing concept grew, the NoSQL database's uses grew as well, because the NoSQL databases, in general, are very easy to scale, which is very useful to the cloud computing environment. Thus, Jakarta EE needed to provide an improvement to provide a common solution to the NoSQL databases.

Jakarta NoSQL is the answer of Jakarta EE to the problems about integration with the NoSQL database and is the first specification to be born inside Jakarta EE. But why not improve Jakarta persistence to attempt to the NoSQL database instead of creating a new specification? To respond to that, we'll need to take a closer look at the NoSQL database concepts and the challenges to work with them. In this book, we'll not dive deep into the NoSQL

database concepts, but we will have a small overview of these concepts only to understand the Jakarta NoSQL goals.

The first thing to know about the NoSQL database is that there is no standard for these databases. This means that each NoSQL database can work in a particular manner and can have its own features that differ from each other. Although NoSQL database does not have a standard, these are classified into four categories, which are as follows:

Column

Document

Key-Value

Graph

Another characteristic between almost all the NoSQL databases is that these do not work with transactions like a relational database, and the developer needs to care of that when the application needs a transaction.

At this point, we can already see a big difference between the NoSQL database and the relational database, and we can understand why it was needed to create a new specification to work with the NoSQL database. Now, we can explain more about the Jakarta NoSQL goals.

Jakarta NoSQL has a goal of providing an API to integrate the Java applications to the different NoSQL databases, with a common API to work with the different types of NoSQL databases. With this, in Jakarta EE, you will have a standard way to integrate with many NoSQL databases. To each category of the NoSQL database, the Jakarta NoSQL will have a module to work with this category. So, Jakarta NoSQL will have the following:

Column communication API

Document communication API

Key-value communication API

Graph communication API

Mapping API

Note that we have the mapping API module. This module is not related to the communication but is a layer based on annotations and CDI to preserve the integration with the other Jakarta EE specifications.

Each module will have its own TCK, which means, we can have a vendor that is totally compliant to one module but is not compliant with another module.

Jakarta NoSQL brings a big advantage to the developers because it creates a standard API built, with the workings of many vendors, promoting a stable solution to the NoSQL database integration. Furthermore, the developers do not need to care about the specific details of the NoSQL database, because Jakarta NoSQL creates an abstraction layer that abstracts the details of the NoSQL database from the developers. This specification is expected for Jakarta EE 10.

Eclipse enterprise for Java

Jakarta EE is under Eclipse Enterprise for Java (EE4J) that is responsible for managing the Jakarta EE brand and for all the projects for creating the standards that will compose Jakarta EE.

EE4J has the EE4J **Project Management Committee** which is responsible for maintaining the overall vision for the top-level project. This project has these companies as an active member – Tomitribe, IBM, Red Hat Webtide, Oracle, Fujitsu, Payara, Liferay, and others. As you can see, this project is supported by many big companies, and it is a big advantage to adopting this solution.

From Java EE to Jakarta EE

The migration for Java EE from Oracle to Eclipse Foundation was not simple, because many questions emerged at this stage, and the community needed a lot of work to respond to these questions in a satisfactory manner. The main question that emerged was the Java name because Oracle gave in the Java EE to the community, but the trademark Java still belonged to Oracle. It meant the Eclipse Foundation or the community could not use Java in the project's names or namespaces, but the Java EE had Java in its name and it had the javax in the namespace. So, the community needed work to solve it and change all the namespace containing javax.

With this, the first change in the migration of Java EE to Jakarta EE was to create a new process to evolve as discussed previously, and to update the namespace from javax.* to jakarta.*. Jakarta EE 8 was released without changing the namespaces because this version only focused on the update of the process. The update in the namespaces was applied on Jakarta EE 9.

Jakarta EE 9

Jakarta EE 9 came with the update of the namespace as the main change. Version 9 is compliant with Java 8 and does not support Java 11 that is the next LTS Java version after Java 8. However, the community has released another version of Jakarta EE compliance to Java 11. This is the Jakarta EE 9.1, where the only change was supported to run with Java 11 version.

Understanding the Jakarta EE Tiers

The Jakarta EE has a distributed multitier application model, but it has three well-known and widely used tiers, which are – presentation tier, business tier, and integration tier. In the past, it was very common to use the presentation tier and the business tier in different packages and a lot of the time went in running the different hosts. Over time, this approach was no longer used and these tiers were put together in the same package inside an EAR project and WAR project. Nowadays, the application that uses this approach, in general, is a legacy project.

Jakarta EE is built based on the components and these components are a self-contained unit that can communicate with each other's components. Each component is related to one tier, although some components work in multiples tiers, [such as Jakarta **Contexts and Dependency Injection** each component is native to one-tier. Thus, the component from the presentation tier is known as **web components** and the components from the business tier are known as **business** *Figure 1.3* illustrates these tiers as follows:

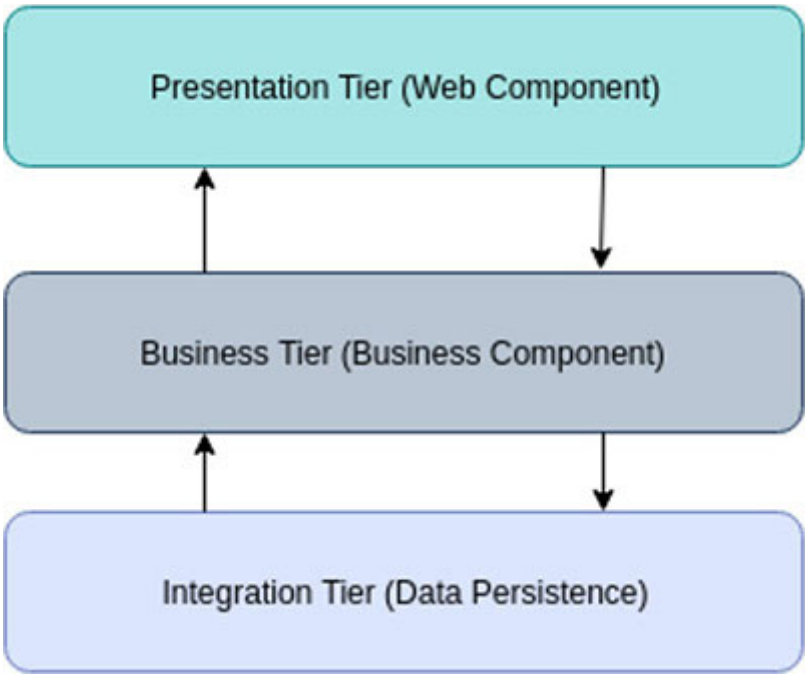

Figure 1.3: *Jakarta EE tiers*

In the presentation tier, we have the specifications for the web, to provide the solutions to the common needs related to the web in the enterprise applications. Some of those needs are as follows:

Communication using the HTTP protocol

Rich pages

Communication using Restful

Communication using WebSocket

Communication using SOAP

We have much more that can be listed, but those are the main and more common needs. All those needs are solved with some specification of the presentation tier. Those are the main specifications of the Jakarta EE's presentation tier, as follows:

Jakarta Servlet

Jakarta Page

Jakarta Faces

Jakarta RESTful Web Services

Jakarta WebSocket

In the business tier, we have specifications to the business, to provide the solutions to the common needs related to the business rules in the enterprise applications. Some of those needs are as follows:

Data persistence

Transaction

Asynchronous processing

Scheduled tasks

All those needs are solved with some specification of the business tier. Those are the main specification of Jakarta EE's business tier, as follows:

Jakarta Enterprise Beans

Jakarta Persistence

Jakarta Transaction

Jakarta Messaging

Some of those specifications will be covered in more detail. This chapter is only an introduction, and we'll not cover any specifications in this chapter.

Besides that, we have some specifications that act in all the tiers, working as a cross-tier specification. The main cross-tier specification is CDI, but we can quote the others like Jakarta Concurrency and Jakarta Security.

In the past, those tiers (the presentation tier and the business tier) were packaged separately and each tier started in a different JVM. The integration between the presentation tier and the business tier was done via a remote EJB. Over time, this approach has changed and those tiers are now packaged together in an EAR project and started in the same JVM. Furthermore, the

remote EJB is used lesser and new techniques for the integration are created. Nowadays, the remote EJB is not needed anymore and is used only in the legacy applications.

Jakarta EE profiles

Jakarta EE works with the concept of profiles, and nowadays, it has two profiles to work on, but in the future, we can see new profiles being born in Jakarta EE. You may ask, what are the Jakarta EE profiles?

The Jakarta EE profiles are a collection of the Jakarta EE technologies and APIs that attend to a specific application type. Thus, the developer can choose the profile that is better for the application and avoid making the imports or starting the Jakarta EE technologies and APIs that won't be used by the application. The Jakarta EE profile are as follows:

Web Profile

Full Profile

The web profile is a profile for the web applications that do not need to use the specification from the business tiers. This profile contains the web technologies and is focused on providing the solutions to the web problems.

The full profile is a profile to create the heavy-weight applications that need to use the specifications from the business tier to solve many problems. This profile contains all the technologies and APIs from Jakarta EE and should be used only when needed.

Note that the Jakarta EE profile and the tiers are not the same, but that it has a relation. When we talk about the tiers, we are talking about how the application is divided and each tier can be in one machine. But when we talk about the Jakarta EE profiles, we are talking about the dependencies needed by the application and what the technologies used by the application are.

Server applications

The Jakarta EE application generally is a system that is accessed by HTTP and makes the integrations with the other applications — Jakarta EE applications or not — and resources. Thus, Jakarta EE has the server application, and the Jakarta EE applications are deployed in a server application to process its tasks and be able to be accessed by the uses and systems. But what does the Jakarta EE's server application have, that is special and different from the other servers?

The server applications in Jakarta EE are servers that implement the Jakarta EE platform API and provides the standard Jakarta EE services. In other words, the server applications provide the implementations of each specification, meeting the TCK tests. Besides that, the server application provides the Jakarta EE container that is the Jakarta EE's heart.

The Jakarta EE container is responsible for providing many abstractions, where the developer does not need to care about many things because the Jakarta EE container already cares for him. As examples of things that can be cared for by the Jakarta EE container are transactions, bean life-cycle, and others.

The Jakarta EE container is divided into three types, which are as follows:

Jakarta EE web container

Jakarta EE application client container

Jakarta EE enterprise beans container

The Jakarta EE web container is the interface between the web components and the webserver, and it promotes many abstractions to work with the web components and web resources. The CDI bean life cycle is managed by this container.

The Jakarta EE application client container is the interface between the Jakarta EE application clients and the Jakarta EE server. This is used less nowadays because it was used more by the Enterprise Beans clients that consumed the Enterprise Beans service.

The Jakarta EE enterprise beans container is the interface between the enterprise beans and the Jakarta EE server. This container manages the executions, life cycle, transaction, and the others from the Enterprise Beans components. Throughout this book, we will see many of these abstractions in practice.

When we start up a server application, the server application starts each Jakarta EE container and resources and makes them available to the deployed applications. Furthermore, the server application manages many questions about clustering a Jakarta EE application and provides the facilities when we want to make a cluster of the Jakarta EE applications.

We have many server applications in the market, provided by many small, medium, and big vendors. As an example of a server application, we can quote Wildfly, GlassFish, Open Liberty, and others. Furthermore, we have the embedded servers like Thurntail, TomEE, and others that are servers embedded in a jar application. These type of servers won't be covered in this book.

Requirements for working with Jakarta EE

Basically, what we need is a JDK installed to compile and run the Java application, an IDE to code, and a server application to deploy, but in this book, we will use Maven to manage the dependencies of the project. Thus, our project will be a maven project, but it is not mandatory and you can create an Eclipse project, Gradle project, or others.

The JDK to create the examples of this book is JDK 1.8 (Java 8) and you can download it on the Oracle website. The Jakarta EE 9.1 supports the JDK 11.

The IDE used by me is IntelliJ, but you can use any IDE as it is a maven project and is decoupled from IDE.

The server application used in all the examples is Wildfly 24, but you can use any server application that is fully Jakarta EE 9 certified. The code shown in this book does not need changes if you use another server, but the way to make the configurations on the server can be different. But this book will not focus on the server configurations.

To start working with Jakarta EE 9, we should create a WAR project or an EAR project and define the Jakarta EE 9 as a dependency on But we highly recommend that you use the WAR

project as the EAR has complexities that are not needed for the current projects.

The following is an example code:

```
1.
2.          ...

8.          ...
9.
```

After you import this project in your IDE, you can start working with Jakarta EE. Thus, you should start the server application and deploy this application into the server application. You can do this by using your IDE if your IDE supports the integration with the server application or you can do this outside IDE. The details of IDE and the server application are not in the scope of this book, but I'm putting it here to help you if required.

Conclusion

Jakarta EE makes the development of the enterprise applications easy, promoting several solutions to the well-known enterprise problems. Besides that, it is a specification-based solutions, that is, it promotes the solutions decoupled from the vendors and we have a solution made with the collaboration of many small, medium, and big companies.

Throughout this chapter, we saw Jakarta EE has a huge experience into promoting the APIs to creating abstractions in the enterprise environment. Furthermore, we saw Jakarta EE now has the vision to use its experience to promote the solutions to a cloud environment. In this new approach, called cloud-native application, we have seen the emergence of many concepts like 12 factors, microservice, serverless, and others. Thus, Jakarta EE will provide many specifications based on these concepts in the upcoming years.

This chapter has exposed an overview of how it works, how its architecture was designed, what are Jakarta EE's tiers, what are the Jakarta profiles, and what is the server application. From the next chapter onwards, we will have the practical contents and we will learn how to use some components of Jakarta EE.

In the next chapter, we will explore the Jakarta Servlet and learn how to use this specification.

CHAPTER 2

Jakarta Servlet

Introduction

In this chapter, we will explain what is Jakarta Servlet, how we can create a Servlet class, how we can filter the requests and responses, what is the Forward and RequestDispatcher feature and how we can use them, how we can work with the asynchronous process in a Servlet, how we can work with Nonblocking I/O, and how we can use the Server Push. This chapter is very practical, and after reading this chapter, you will be able to create Servlets in your Jakarta EE applications using the best practices to it.

Structure

In this chapter, we will discuss the following topics:

Understanding what is Jakarta Servlet

Creating a Servlet

Filtering the Request and Response

Understanding RequestDispatcher

Asynchronous processing

Using Nonblocking I/O

Using server push

Objectives

The main objectives of the chapter are to enable the readers to use the Jakarta Servlet in a better way to work with Servlets in a synchronous and asynchronous way. Besides that, the reader will be able to use Nonblocking I/O and Server Push with Servlet.

Understanding what is Jakarta Servlet

The Jakarta Servlet is a technology that provides support to the network communications in a request response way. In other words, Servlet permits communications where a requester applies a request to the server and the server sends back a response. It can support any protocol, but the Servlet specification requires the HTTP support because HTTP is the most commonly used protocol. With this, in this book, we will only work with Servlet over HTTP.

In the Jakarta Servlet, we create a Servlet class that is responsible to process the requests and send a response to the requester. Each Servlet class is responsible to process the request according to the listener configuration. A listener configuration can be provided by annotation or by deployment descriptor Throughout this chapter, we'll see what a listener configuration is and how can we create them. For now, you don't need to think about it.

A Servlet class extends the **jakarta.servlet.http.HttpServlet** class and has the methods to each HTTP method, and we can override them when we want to insert a behavior to a respective HTTP method request. The following are the methods:

doGet(HttpServletRequest req, HttpServletResponse resp)

doHead(HttpServletRequest req, HttpServletResponse resp)

doPost(HttpServletRequest req, HttpServletResponse resp)

doPut(HttpServletRequest req, HttpServletResponse resp)

doDelete(HttpServletRequest req, HttpServletResponse resp)

doOptions(HttpServletRequest req, HttpServletResponse resp)

doTrace(HttpServletRequest req, HttpServletResponse resp)

Besides that, the Servlet class has the service method that is the method called when a request is received, and this method calls one of the methods described earlier, according to the request's HTTP method. Generally, when we create one code to be executed in all the requests independently from the HTTP method, we override the service method, writing the logic code there. We will see it soon.

Servlet lifecycle

A Servlet has a well-defined life cycle that is managed by the Jakarta EE container; thus, the developer does not need to care about the life cycle. Besides that, the developer can write some actions or behaviors to be executed in each step. Thus, when a request is mapped to some Servlet, the Jakarta EE container executes the following steps:

If an instance of the servlet class does not exist, then the container creates the instance. The servlet can be started either when the container is started or when the container detects that the servlet is needed.

The servlet is initialized by calling the **init** method. In case the developers want to insert some actions or behaviors at initialization time, he can override the **init** method.

After the servlet was initialized, the servlet is able to treat the requests. When a request is sent, the **service** method is called which is responsible for handling the request. The service method receives the **ServletRequest** and **ServletResponse** instances that represent the request and response respectively. In a the **service** method delegates the execution to a HTTP method or others described earlier) according to the HTTP protocol. If the developer wants to write one action for all the HTTP methods, he can override the **service** method.

When the servlet is removed, the **destroy** method is called. Thus, if the developer wants to write one action to be executed at the remove time, he can override the *destroy* method.

These are the steps of the servlet life cycle, and each step is executed by the Jakarta EE web container.

Multithreading

The Servlet should be able to respond to multiple concurrent requests sent to them. To do it, the Jakarta EE web container creates one thread for each request and these threads share the objects stored in the Servlet context, objects stored in the HTTP session, and any resource that is in the instance variables of the servlet. The resources, objects, or values that are inside the request or response instances are not shared between the requests. Thus, the developer should pay attention to the questions about concurrency to avoid the side effects. Therefore, if we create or access an instance variable of the servlet inside any of the HTTP methods of the servlet mentioned earlier, such as **doGet** and we must be aware that the very same servlet instance is reused between all the requests hitting the servlet. As a tip for the best practice, avoid assigning the request or session scoped variables as the instance variables of the servlet.

Creating a Servlet

To create an example of servlet, first we'll create a new project called *sampleservlet* using the archetype described in *Chapter 1, Introduction to* Thus, to create a project, you can use the Maven command as follows:

```
mvn archetype:generate -DarchetypeGroupId=net.rhuanrocha -
DarchetypeArtifactId=jakartaee9-war-archetype -
DarchetypeVersion=1.0.SNAPSHOT -DgroupId=net.rhuanrocha -
DartifactId=sampleservle
```

Note that you can choose another **-DgroupId** and/or

As the first example, we'll create a servlet that will receive a GET HTTP request and will respond to a simple message like The following is the **HelloWorldServlet** class:

```
1. package net.rhuanrocha.sampleservlet;
2.
3. import jakarta.servlet.ServletException;
4. import jakarta.servlet.http.HttpServlet;
5. import jakarta.servlet.http.HttpServletRequest;
6. import jakarta.servlet.http.HttpServletResponse;
7. import jakarta.io.IOException;
8.
9. public class HelloWorldServlet extends HttpServlet {
```

```
10.
void req, HttpServletResponse resp) throws ServletException,
IOException {
12.
14.     }

15.}
```

The **resp.getWriter** returns the **OutputStream** used by the servlet to write to the browser.

Note that the class should extend the **HttpServlet** class, and as we only want to respond to the **GET** method, we have overridden the **doGet** method. But this class will respond to which URI? At this time, this servlet class will not respond to any URI because we should tell the Jakarta EE web container about which URI the servlet will respond to. As described in *Chapter 1, Introduction to Jakarta* this can be done in two ways – through the deployment descriptor or the annotations. This book will focus on the use of annotations; however, this time we'll use the deployment descriptor only to demonstrate how to do it by using one. To do that, we should create the **web.xml** inside the **WEB-INF** folder The following is an example of

```
1. version="1.0"
2.              encoding="UTF-8"?>
3. >
12.

13.
```

To explain more about the first we should declare the servlet defining a unique name and telling the servlet class. Look at the following code snippet:

1.

4.

Then, we need to map the URL to the declared servlet. This is the URL that the servlet will respond to. It is done by the **url-pattern** tag. Look at the following code snippet:

1.

In this case, the servlet will respond to As we are using the Wildfly, the following is the URL to test:

http://localhost:8080/sampleservlet-1.0.SNAPSHOT/helloworld

To test it using curl, you can execute it as follows:
curl http://localhost:8080/sampleservlet-1.0.SNAPSHOT/helloworld

The following is the return:
Hello World.

As you can see, the servlet's output is

Configuring the Servlet using annotation

Since Servlet 3.0, we can use the **@WebServlet** annotation to make the configurations without the Then, we need to annotate the servlet class. Look at the following example, where we will create a similar servlet class, but now we'll configure using the annotation and the **web.xml** will not be created:

```
net.rhuanrocha.sampleservlet;
2.
jakarta.servlet.ServletException;
jakarta.servlet.annotation.WebServlet;
jakarta.servlet.http.HttpServlet;
jakarta.servlet.http.HttpServletRequest;
jakarta.servlet.http.HttpServletResponse;
jakarta.io.IOException;
9.
10. = value )
class HelloWorldServlet extends HttpServlet {
12.
13.        protected void req, HttpServletResponse resp) throws
ServletException, IOException {
14.
15.
16.
17.        }
18. }
```

Note that the only change applied to the servlet class is the annotation, that defines a **servlet-name** and

Like the preceding example, the servlet will respond to As we are using the Wildfly, the following is the URL to test:

http://localhost:8080/sampleservlet-1.0.SNAPSHOT/helloworld

To test it using you can execute it as follows:

curl http://localhost:8080/sampleservlet-1.0.SNAPSHOT/helloworld

The following is the return:

Hello World.

The **@WebServlet** has other parameters, which are as follows:

Declares the servlet name.

Declares the URL patterns of the servlet. Through it, the servlet will be mapped to the URL.

Declares if the servlet supports the asynchronous operations.

Declares a description of the servlet.

Declares the servlet name to display.

Declares the **init** parameters of the servlet.

Declares the large-icon of the servlet.

Defines an order to start the servlets.

Declares the small-icon of the servlet.

Declares the URL patterns of the servlet. Through it, the servlet will be mapped to the URL.

Note that **urlPatterns** and **value** are similar. One of them should be declared but not both together. If you configure one or more URL patterns using either **urlPattern** or it will work better. However, it is recommended to use **value** when you have a single URL pattern and use **urlPatterns** when you have multiple URL patterns.

URL Patterns

The URL patterns defining each URL will be executed by the servlet. If a URL matches the servlet's URL patterns, then this Servlet will process the request. In the servlet created earlier, we configured the URL pattern to Thus, all the requests made to will be processed by the **HelloWorldServlet** class. The following are some examples of accepted patterns, as shown in *Table*

Table 2.1: *Servlet URL Patterns*

Request parameters

The request parameters are string values sent from the client-side to the servlet. These parameters can be sent by both, the URI query string and the POST-ed data. The request parameters are stored as a map of the name-value pairs inside the **HttpServletRequest** object. And you can handle it by the following methods:

getParameter

getParameterNames

getParameterValues

getParameterMap

At this point, we'll create a new servlet to show you how we can get the parameters sent on request. In our example, we will create a servlet called **DNSServlet** that will have a **POST** method to persist a new mapping of the domain name to IP. Later, we will create a **GET** method to read these IPs according to the domain name. You can see the **DNSServlet** class as follows:

```
1.  package net.rhuanrocha.sampleservlet;
```

```
2.
jakarta.servlet.ServletException;
jakarta.servlet.annotation.WebServlet;
jakarta.servlet.http.HttpServlet;
jakarta.servlet.http.HttpServletRequest;
jakarta.servlet.http.HttpServletResponse;
java.io.IOException;
java.util.Objects;
java.util.concurrent.ConcurrentHashMap;
java.util.concurrent.ConcurrentMap;
12.
13. = "DNS urlPatterns =
class DNSServlet extends HttpServlet {
15.
16. // Map to store the IP mapped by domainName
17.         private ipMap;
18.


19.         public void throws ServletException {
20.                 ipMap = new
21.         }
22.
23.         public void req, HttpServletResponse resp) throws IOException
{
24.
25.                 String domainName =
26.                 String ip =
27.
28.                 //In case of any value be null a 400 HTTP status is
returned to client.
29.                 == null || ip == null){
30.
```

```
31.
32              }
33.
34.              ipMap.put(domainName,ip);
35.
36.              //A 201 HTTP status is returned to client
37.
38.
39.         }
40.
41.  }
```

In the preceding example, we can see a lot described throughout this chapter. The first part is the instance variable created in the class. Look at the following code snippet:

```
// Map to store the IP mapped by domainName
private ConcurrentMapString> ipMap;
```

As you can see, it is a As described earlier, an attribute inside a servlet class is shared between all the requests processed by an instance of this class. So, we should use a thread-safe map to avoid the side effects.

Later, we override the **init** method that is called as soon as the servlet object instance is created.

When we receive the request, we get the parameters using **getParameter** and passing the parameter name to the method.

Look at the following code snippet:

```
String domainName = req.getParameter("domainName");
String ip = req.getParameter("ip");
```

As you can see, you get the **domainName** and **ip** parameters from the **HttpServletRequest** object. To make the tests in this servlet, you can do it.

Like the preceding example, the servlet will respond to

Note that based on all the request with **prefix /dns** will be matched to the **DNSServlet** class. The following is another valid URL:

POST:

As we are using the Wildfly in localhost, the following is the URL to test:

POST: **http://localhost:8080/sampleservlet-1.0.SNAPSHOT/dns**

To test it using curl, you can execute it as follows:

```
curl http://localhost:8080/sampleservlet-
1.0.SNAPSHOT/helloworldcurl -d
"domainName=rhuanrocha.net&ip=127.0.0.1" -X POST
http://localhost:8080/sampleservlet-1.0.SNAPSHOT/dns -v
```

The following is the return:

```
> POST /sampleservlet-1.0.SNAPSHOT/dns HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 38
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 201 Created
< Connection: keep-alive
< Content-Length: 0
```

Now we'll create the **doGet** method to process the requests using the **GET** method. In this method, we will search an **IP** by the domain name. Look at the method **doGet** as follows:

```
void req, HttpServletResponse resp) throws ServletException,
IOException {
2.
3.          String domainName =
4.
5.          //In case of domainName be null a 400 HTTP status is
returned to client.
6.          == null){
7.
resp.sendError(HttpServletResponse.SC_BAD_REQUEST);
8.
9.          }
```

```
10.
11.             PrintWriter printWriter = resp.getWriter();
12.             printWriter
13.                     .println(Optional
14.
15.
16.
17.
18.         }
```

As you can see, we got the parameter and then we got the **IP** from **ConcurrentMap** and printed it for the client. The following is the complete **DNSServlet** after we created the

```
1.  package net.rhuanrocha.sampleservlet;
2.
jakarta.servlet.ServletException;
jakarta.servlet.annotation.WebServlet;
jakarta.servlet.http.HttpServlet;
jakarta.servlet.http.HttpServletRequest;
jakarta.servlet.http.HttpServletResponse;
java.io.IOException;
java.io.PrintWriter;
java.util.Objects;
11. import java.util.Optional;
12. import java.util.concurrent.ConcurrentHashMap;
13. import java.util.concurrent.ConcurrentMap;
14.
15. = "DNS urlPatterns =

16. public class DNSServlet extends HttpServlet {
```

17.
Map to store the IP mapped by domainName
ipMap;
20.
void throws ServletException {
23. ipMap = new
24. }
25.
26. @Override
void req, HttpServletResponse resp) throws IOException {
28.
domainName =
ip =
31.
case of any value be null a 400 HTTP status is returned to client.
== null || ip == null){
36. }
37.
38. ipMap.put(domainName,ip);
39.
201 HTTP status is returned to client

42.
43. }
44.
45. @Override
void req, HttpServletResponse resp) throws IOException {
47.
domainName =
49.
case of domainName be null a 400 HTTP status is returned to
client.

```
== null){
54.             }
55.
printWriter = resp.getWriter();
57.             printWriter
61.
62.
63.     }
64.
65.}
```

To test it using curl, you can execute it as follows:

curl http://localhost:8080/sampleservlet-1.0.SNAPSHOT/dns?domainName=rhuanrocha.net

The following is the return:

127.0.0.1

As you can see, the output is

HTTP Headers

The HTTP has the HTTP headers that can be sent from the client to the server on the HTTP request or can be sent from the server to the client on the HTTP response. To access the headers on an HTTP request, you can use the following methods:

getHeader

getHeaders

getHeaderNames

Look at the following example where we get the header called

String contentType = req.getHeader("Content-Type");

If you want to put some HTTP header to response, you can use the following methods:

setHeader

addHeader

Look at the following example where we set the header called

resp.setHeader("Content-Type","text/html; charset=UTF-8");

Note that to get the HTTP header from a request, we use **HttpServletRequest** and to set HTTP header to the response, we use

Managing session

A common task when we are working with the HTTP protocol is to manage the session to put some data that will survive for the many requests. Basically, when the client starts the access to any HTTP server, an HTTP session is created in the HTTP server; this session is kept for many or all requests done by the client. The Servlet provides the **HttpSession** interface that is used to manage the HTTP session. So, you can invalidate a session to create another one or put some data to it. Note that the HTTP session has a time out, and after a time out happens, the session is invalidated and a new session is created. The following are the main methods of the **HttpSession** interface:

getSession

getId

getCreationTime

invalidate

getLastAccessedTime

setAttribute

getAttribute

To get the session associated to the request call, the following method of **HttpServletRequest** is used:

HttpSession session = req.getSession();

In the preceding code snippet, the current session will be returned or null in case there is no session associated. If you want a new session to be created in case there is no session associated, you can call the following method of

HttpSession session = req.getSession(true);

To put some data to session, you can use this method of

session.setAttribute("user", user);

In the preceding code snippet, we are setting the user as an attribute and it is associated with the key. So, to get the user from the session, you can call the following method from

session.getAttribute("user");

This method returns an attribute from the session according to some key.

Filtering the request and response

Many a times, we want to apply some transformations in the data and headers of an HTTP request or HTTP response and/or make any validation before the request is executed. A good approach to do it is using the filters, where we filter the requests and apply the transformations. Using this approach, you can decouple the request processing logic from the transformation logic, and reuse the transformation logic in many servlets. The following are some scenarios that we can be filters:

Logging and auditing

Authentication

Tokenizing

Data compression

Image conversion

Decrypting data

It is nice! But how can we use this approach in a servlet?

The Jakarta Servlet has a feature to create the filter classes to intercept the requests and responses and apply some logic to this. Basically, a **filter** class is a class that implements the **Filter** interface and uses the **FilterChain** and **FilterConfig** to handle how the filter should work. The Servlet Filters is based on the pattern called **Chain of** that is a behavioral pattern of **Gang of Four** The following is an example of a Servlet Filter to filter a request:

```
jakarta.servlet.*;
java.io.IOException;
java.util.HashMap;
java.util.Map;
java.util.Objects;


6.
7. = urlPatterns = "/ 1.
class AuthorizationFilter implements Filter {
9.
10.        private MapString> passwords;
11.        @Override
12.        public void filterConfig) {
13.
14.              passwords = new ConcurrentMap<>();
15.
16.
17.
18.        }
19.
20.        @Override
21.        public void servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
22.
```

```
servletRequest,(HttpServletResponse) servletResponse)) {
24.                 ((HttpServletResponse) servletResponse).
sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorided
access
25.             }
26.


27.                 filterChain.doFilter(servletRequest, servletResponse);
28.             }
29.
30.
31.     }
32.
boolean request, HttpServletResponse response){
34.
35.             String login =
36.
(login == null || !passwords.containsKey(login) ){
39.             }
40.
41.             String password =
42.
(password == null || !passwords.get(login). equals(password) ){
45.             }
46.
48.
49.     }
50.}
```

In the preceding example, we created a Servlet Filter to validate the authorization based on a login and password. The login and password information are coming in the HTTP header. Note that it is only an example to explain the Servlet Filter feature, and sending the password through the HTTP header is not recommended, mainly if you are not using HTTPS.

The Servlet Filter can be configured by **web.xml** or by annotation. In our example, we used annotation. Look at the following code snippet:

@WebFilter(filterName = "AuthorizationFilter", urlPatterns = "/security/*")

As you can see, we configure a filter name and URL pattern. Based on the **urlPattern** attribute, all requests done to **http://{host}:{port}/{context-root}/security** will be intercepted by this filter that will make validation about authorization.

Furthermore, we override the method that is executed when an instance **AuthorizationFilter** is created. In this method, we initialize the password's map, putting two logins and passwords that will be used for validation. Look at the following code snippet:

```
1.    @Override
void filterConfig) throws ServletException {
3.
4.          passwords = new ConcurrentMap<>();
7.
8.    }
```

Then, we have the **doFilter** method that will be called when the request and response are intercepted. Note that you can have many filters to the same request/response and when we call the **filterChain.doFilter(servletRequest,** we are calling the next filter or the servlet, if it no longer has a filter. If the **filterChain.doFilter(servletRequest, servletResponse)** is not called, then neither the next filter or the servlet will be called. Look at the following code snippet of

```
1.  @Override
2.      public void servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {
3.
4.          servletRequest,(HttpServletResponse) servletResponse)) {
5.              ((HttpServletResponse) servletResponse).
sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorided
access
6.          }
7.
8.              filterChain.doFilter(servletRequest, servletResponse);
9.          }
10.
11.
12.     }
```

Understanding Forward and RequestDispatcher

The **RequestDispatcher** is a useful feature that provides support to forward or include another resource to make a response. Basically, the Servlet has an object from **RequestDispatcher** and it has the methods, **include** and to call these functionalities.

Forward

The **forward** is used when we want to transfer the control of the processing to another web component to generate the response. In other words, the responsibility to generate a response is of the servlet forwarded.

Include

The **include** is used when we want to include another web component in the response, but the control of the processing still is of the servlet that makes the include.

When we apply a forward or include, the attributes of the request is shared between the web components. So, the attributes will be exactly the same. Look at the following example where we have a servlet that makes an include or forward according to the parameter sent:

```
1. import jakarta.servlet.RequestDispatcher;
```

```
2. import jakarta.servlet.ServletException;
3. import jakarta.servlet.annotation.WebServlet;
4. import jakarta.servlet.http.HttpServlet;
5. import jakarta.servlet.http.HttpServletRequest;
6. import jakarta.servlet.http.HttpServletResponse;
7. import java.io.IOException;
8. import java.io.PrintWriter;


9.
10.
=
12. public class DispatcherServlet extends HttpServlet {
13.
void req, HttpServletResponse resp) throws IOException {
17.
type =
18.
writer = resp.getWriter();
24.
dispatcher =
27.                 dispatcher.forward(req, resp);
{
29.                 dispatcher.include(req, resp);
30.             }
33.     }
34.
35.


36.}
```

As you can see, we made a dispatcher to the servlet attending at that is a servlet shown previously. Look at the following code snippet:

RequestDispatcher dispatcher = req.getRequestDispatcher("/helloworld");

Note that in this servlet, I override the service method. So, all the requests will be processed by the service method, independently of the HTTP method used.

To illustrate the difference between forward and include, I will apply two tests, one passing and the other passing Look at the following example:

Forward:

Request:
curl http://localhost:8080/sampleservlet-1.0.SNAPSHOT/helloworld.html?type=forward

Response:
Hello World.

Include:

Request:
curl http://localhost:8080/sampleservlet-1.0.SNAPSHOT/helloworld.html?type=include

Response:

Hello World.

Note that in the print generated by **DispatcherServlet** was not sent to the client. However, when we use the all the content is appended and sent to the client, according to the call order.

Note that we printed the HTML, but it is not a good practice if you want to return an HTML, instead, the JSP is used; however, this book does not talk about the JSP.

Asynchronous processing

In the Jakarta EE platform, each request done to a Servlet is processed by a thread. These threads stay in a pool of container threads and after processing a request, it is returned to the pool of container threads. With this, the request processing needs to be very fast to release the thread to another request processing. It is very important, because it can boost or degreed the application performance a lot according to the use of these threads. But, sometimes we need a long-term process to execute some kind of hard process. In these cases, we can use the asynchronous processing disposed on the Jakarta Servlet.

The asynchronous processing in the servlet permits us to delegate the execution to an asynchronous process managed by the context that will generate the response to the client. Note that the client will keep waiting to respond, but the difference is that the execution will happen in another thread, permitting the servlet thread to be released to execute another request. As an example, we will create a servlet class called that will emulate the behavior of creating reports. Look at the following code:

```
1. import jakarta.servlet.AsyncContext;
2. import jakarta.servlet.ServletException;
3. import jakarta.servlet.annotation.WebServlet;
4. import jakarta.servlet.http.HttpServlet;
5. import jakarta.servlet.http.HttpServletRequest;
6. import jakarta.servlet.http.HttpServletResponse;
```

```
7.  import  java.io.IOException;
8.  import  java.io.PrintWriter;
9.  import  java.util.Optional;


10.
= =
12.  public  class  ReportServlet  extends  HttpServlet  {
13.
void  resp)  throws  IOException  {
15.
AsyncContext  asyncContext  =  req.startAsync();
18.
19.              asyncContext.start(new  {
void  run()  {
name  =  Optional
23.                                        .ofNullable(asyncContext.getRequest().
25.
{
printWriter  =  asyncContext.  getResponse().getWriter();


33.
34.                        }  catch  e)  {
35.                              e.printStackTrace();
36.                        }
37.                        finally  {
38.                              asyncContext.complete();
39.                        }
40.
41.
42.                   }
43.              });
44.
```

```
45.
46.     }
47.
String name){
thread to simulate a long-term process to generate report
{
52.             } catch e) {
53.                     e.printStackTrace();
54.             }
55.
new


61.                             .append(name)
64.     }
65.}
```

As you can see, the **generateReport(String name)** is emulating a slow process to generate a report.

First of all, we need to configure the servlet to accept the asynchronous processing. It can be done by setting the **asyncSupported** attribute to the true value. Look at the following code snippet:

```
@WebServlet(value = "/report",asyncSupported = true)
```

Then, we need to start the asynchronous processing:

```
final AsyncContext asyncContext = req.startAsync();
```

Later, we just start the thread using the **AsyncContext** class. Note that we have set the timeout to 90000 milliseconds, because in general, the server application has a default timeout of 10 seconds.

If the servlet is filtered by any servlet filter, we need to set the **asyncSupported** attribute of the filter to the true value as well. Look at the following example:

@WebFilter(filterName = "HelloWorldFilter", urlPatterns = "/*", asyncSupported = true)

The following is an example of a test using

Request:
curl http://localhost:8080/sampleservlet-1.0.SNAPSHOT/report

Response after 1 minute:

name:

With this, the server responds to the client without blocking the servlet thread for a long time. Note that the client will not see any difference.

Using Nonblocking I/O

Asynchronous processing is a good servlet feature to improve the performance of the servlet threads and permit a thread to be available to respond to the other requests. But, imagine you are receiving a request and the network is very slow. The receiver will read the data faster than the requester can write and the result will be a momentarily sitting idle of the receiver. Thus, even with the asynchronous processing, the servlet thread will be kept for a long time and will decrease the performance. Note that we are talking about the input/output stream between the client and the server.

To solve this problem, the servlet has the Nonblocking I/O feature that permits reading and writing the data asynchronously to a listener when the data is available. This mechanism creates a separated thread-safe to read and write the data. As a result, the servlet thread responsible for receiving the requests won't wait for the read/write to finish to be available for the other requests.

To use the Nonblocking I/O, you should enable the **async** request processing in the servlets and filters, like in the asynchronous processing. The following is an example of Nonblocking IO:

```
1. import jakarta.servlet.*;
2. import java.io.IOException;
3. import java.util.ArrayList;
4. import java.util.List;
```

```
5. import java.util.Optional;
6. import java.util.logging.Logger;
7.
8.


= =
10. public class ReportNonblockingServlet extends HttpServlet {
11.
static Logger logger = Logger.
13.
void req, HttpServletResponse resp) throws ServletException,
IOException {
15.
AsyncContext asyncContext = req.startAsync();
17.
to InputStream
ServletInputStream inputStream = req. getInputStream();
ReadListener() {
21.
buffer[] = new *
23.                 StringBuilder data = new StringBuilder();
24.
void throws IOException {
27.
{
{
length = inputStream.read(buffer);


String(buffer, length));
32.                         } while (inputStream.isReady());
33.                     } catch (IOException ex) {
34.                         logger.severe(ex.getMessage());
```

```
35.                     }
36.                 }
37.
void throws IOException {
40.
41.                     logger.info(data.toString());
42.                 }
43.
void throwable) {
46.
47.                 }
48.             });
49.
to OutputStream
51.             ServletOutputStream outputStream =
resp.getOutputStream();
WriteListener() {
void throws IOException {
55.                     String name = Optional

56.                             .ofNullable(asyncContext.getRequest().
58.
59.                     List datas = new ArrayList<>();
62.                     datas.add(generateReport(name.toString()));
&& !datas.isEmpty())
66.                     {
68.
69.                     }
70.                     asyncContext.complete();
71.                 }
72.
void throwable) {
```

```
75.
76                  }
77.             });
78.


79
80.       }
81.
String name){
thread to simulate a long-term process to generate report
{
86.             } catch (InterruptedException e) {
87.                 e.printStackTrace();
88.             }
89.
new
95.                         .append(name)
98.       }
99.}
```

In the preceding example, we created the **ReadListener** instance with the **onDataAvailable** method that is executed when there is some data available to read, the **onAllDataRead** method that is executed when all the data is read, and the **onError** method that is only executed when some errors happen. Furthermore, we created the **WriteListener** instance as well, that has the **onWritePossible** method to execute when it has some data available to write and the client is able to receive, and the **onError** method that is only executed when some errors happen.

Note that in the **onDataAvailable** method, we created a **List** of Strings. Look at the following code snippet:

```
1.                  List datas = new ArrayList<>();
```

It permits us to send the data little by little according to the receiver's capacity. Thus, we consume the list. Look at the following code snippet:

```
1.                  && !datas.isEmpty())
2.                   {
4.
5.                   }
```

The following is an example of the test using

Request:
curl http://localhost:8080/sampleservlet-1.0.SNAPSHOT/nonBlockingReport

Response after 1 minute:

name:

Using Server Push

The HTTP/2 has a feature called **Server Push** and the servlet permits you to use this feature from HTTP/2. But what is this feature?

Imagine you have many static contents (images, css, javascript) that you want to render in an HTML page in the browser. These contents don't need to be built by some processing. With this, we can anticipate these contents to the browser while we are processing the remaining content to respond. But how can I do that? Server Push is the answer.

Server Push is a mechanism to anticipate some contents that don't need to wait for the processing to finish because these contents are already formed and will not change according to the processing. In general, it is used for the assets used by the HTML pages. It helps the browser to render the contents with high performance. The following is the **LogoJakartaServlet** as an example of a servlet using HTTP/2:

```
1. package net.rhuanrocha.sampleservlet.servlets;
2.
3. import jakarta.servlet.ServletException;
4. import jakarta.servlet.annotation.WebServlet;
5. import jakarta.servlet.http.HttpServlet;
6. import jakarta.servlet.http.HttpServletRequest;
7. import jakarta.servlet.http.HttpServletResponse;
```

```
8. import jakarta.servlet.http.PushBuilder;
9. import java.io.IOException;
10. import java.io.PrintWriter;
11. import java.util.Objects;
12.


= value )
14. public class LogoJakartaServlet extends HttpServlet {
15.
16.      protected void req, HttpServletResponse resp) throws
IOException {
17.
pushBuilder = req.newPushBuilder();
HTTP is disabled the PushBuilder is returned null.
(pushBuilder != null) {
21.                  pushBuilder
22.
25.                        .push();
26.            }
27.
printWriter = resp.getWriter();
src='images/jakarta_ee_logo.
34.
35.      }
36.}
```

Note that if the HTTP/2 is disabled in the server or in the browser, the **newPushBuilder** method will return null. In general, the servers are configured with HTTP/2 only for the HTTPS communication. Besides that, the application should be running

using Java 8 or major, and it should have the ALPN on classpath because HTTP/2 requires a TLS stack.

You can enable the HTTP/2 to be used in the HTTP communication. As in this book, we are using Wildfly, you can use the following command:

/subsystem=undertow/server=default-server/http-listener=default:write-attribute(name=enable-http2,value=true)

If you are using another server application, check how to enable HTTP/2 to the HTTP communication in the documentation.

To configure the ALPN, you can download it in **http://central.maven.org/maven2/org/mortbay/jetty/alpn/alpn-boot/** and copy it to **$WILDFLY_HOME/bin** and put this command in as follows

```
JAVA_OPTS="$JAVA_OPTS -
Xbootclasspath/p:$WILDFLY_HOME/bin/alpn-
boot-$ALPN_VERSION.jar"
```

Note that it is based on Wildfly as well.

Thus, when the HTTP/2 is enabled, the image will be anticipated to facilitate the rendering. Look at the following code snippet that applies the Server Push:

1. *pushBuilder*

Conclusion

The Jakarta Servlet is a good specification to permit the applications to communicate using many protocols, but mainly the HTTP protocol. This specification is very important, because many other frameworks came based on it to make the communication using the HTTP protocol. As an example of the frameworks that used this specification, we can quote Spring, Struts, VRaptor, and the others. The Jakarta Servlet is now very evolved, and we can use many interesting features described throughout this chapter, such as, Filters, Asynchronous Processing, NonBlocking/IO, and Server Push.

You are now able to use the principal features of Jakarta Servlet in your project to promote the performance solutions to it. The project with all the examples shown in this chapter, stay in

In the next chapter, we will explore the Jakarta Context and Dependency Injection specification, and we'll learn how to use this specification using the best practices.

CHAPTER 3

Jakarta Context and Dependency Injection

Introduction

In this chapter, we will learn what is Jakarta **Context and Dependency Injection** how can we create and inject the beans using CDI, how does the CDI life cycle work, how can we use the producer method and producer field to the product class's instances, how can we use the dispose method, how can we create a CDI interceptor, how can we create a CDI decorator, and how can we use the CDI event.

Structure

In this chapter, we will discuss the following topics:

Understanding Jakarta Context and Dependency Injection

Creating and injecting beans

CDI life cycle

Using producer method, producer field, and dispose method

Creating a CDI interceptor

Creating a CDI decorator

Using CDI event

Objectives

In this chapter, the main objectives are to enable the readers to use the Jakarta Context and Dependency Injection in a better way to create and inject beans, and to create producer methods, producer fields, and dispose methods. Furthermore, the reader will be able to create a CDI interceptor, CDI decorator, and use the CDI event.

Understanding Jakarta Context and Dependency Injection

When we develop the applications using Java, we work with classes and instances. With this, the developer should split the logic, such as each class should have one responsibility. But it is very common that an instance needs a result of executing the logic of an instance of another class to do its work. With this, it is very common that one class has a dependency on another class. In other words, one instance of class will have instances of another class or another. Look at the following example:

Developer class:

```
1. public class Developer {
2.
Brain brain;
4.
6.              brain = new Brain();
7.      }
8.
void problem){
10.
before reasoning
12.             Object solve = brain.reason(problem);
after reasoning
14.     }
15.}
```

Brain class:

```
1. public class Brain {

2.
Object problem){
4.
the reason logic
new
7.      }
8.}
```

Note that the class **Developer** has a dependency of class **Brain** to execute the logic, but in the preceding example, the class **Developer** instantiates Thus, if we need to change the type of the instance from **Brain** to another type, we should update the **Developer** class because besides the class **Developer** instantiating the **Brain** class, **Brain** is not an abstraction (interface or abstract class).

Dependency injection permits us to remove the necessity of the class that has dependency care about how the dependency will be provided. Thus, the class that has the dependency does not know about how this dependency will be provided; they only know that it will be provided from another point. It's very good because it promotes the decoupling of these classes. Look at the following example of dependency injection without CDI:

```
1. public class Developer {
2.
Brain brain;
```

```
4.
brain){
= brain;
7.      }
8.


void problem){
10.
before reasoning
12.             Object solve = brain.reason(problem);
after reasoning
14.      }
15.}
```

The Jakarta **Context and Dependency Injection** permits us to use a dependency injection pattern, but it is not the only feature of this specification. The CDI permits us to create and inject the instances of beans with a well-defined lifecycle according to a scope used, and these instances of beans are managed by context. Furthermore, CDI has a typesafe dependency injection mechanism that permits us to select the dependencies at either the development or the deployment time in an easy way.

CDI has features to permit us to decorate an injected instance, use interceptors, and work with events, and it integrates with many Jakarta components. At this point in the chapter, you do not have to worry about these features, because we'll show you these features in more detail throughout this chapter. In the following code snippet, we have an example of the dependency injection using CDI:

```
1. public class Developer {
2.
Brain brain;
5.
void problem){
7.


before reasoning
9.          Object solve = brain.reason(problem);
after reasoning
11.     }
12.}
```

Note that we used the **@Inject** that is a CDI annotation used to inform the Jakarta context to solve the dependency. Thus, the context will select the implementation that matches the dependency. Both the **Developer** and the **Brain** are managed by the context. The **@Inject** only works when used inside the instances managed by the context. If the instance is not managed by the context — instance was instantiated by a new constructor or another mechanism that is not CDI — the dependency will not be injected.

CDI Scope

When we create an instance, this instance gets a lifecycle. In CDI, we can configure the scope of instances that defines their lifecycles. The following are the CDI scopes and its lifecycles in a Jakarta EE web container:

This scope defines that the lifecycle of the instance is delimited by the HTTP request, that is, the instance will not exist in the next HTTP request. It is defined by the **@RequestScoped** annotation.

This scope defines that the lifecycle of the instance is delimited by the HTTP session, that is, the instance will exist while the HTTP session exists and when the HTTP session is invalidated, all the instances with this HTTP session are dead. It is defined by the **@SessionScoped** annotation.

This scope defines that the lifecycle of an instance is delimited by the lifecycle of the application, that is, the instance will exist while the application is running. It is defined by the **@ApplicationScoped** annotation.

This scope defines that the lifecycle of an instance is delimited by the lifecycle of the client (bean that injects), that is, the instance will exist while the client exists. It is a default scope, but it can be defined by the **@Dependent** annotation.

This scope defines that the lifecycle of an instance is delimited by the developer, which can tell the context when that instance can be marked to die. It is defined by the **@ConversationScoped** annotation.

Look at the following example of how to define a scope to an instance:

```
1.  @ApplicationScoped
class Brain {
3.
4.      public Object problem){
5.
6.          //Execute the reason logic
7.          return new
8.      }
9. }
```

In the preceding example, the instance will exist while the application is running.

Qualifiers

When we are developing an application, it is very common to create multiple implementations of a type (class or interface). Thus, we can create a bean type and have many implementations of this bean type. As an example, we can think about a class called and this class can have many implementations like dog, cat, or others. But using this scenario, if we inject the superclass how will the CDI define the correct implementation? To solve it, the CDI has created the qualifiers.

The qualifier is an annotation that is applied to the bean that helps the developer be able to select the implementation using the qualifier. With this, the developer creates an annotation to each bean implementation — these annotations are the qualifiers — and annotates the beans with the respective annotation (qualifier). Look at the following examples:

The qualifier of Dog implementation is as follows:

```
static
static
static
static
static
static
10.
11. @Qualifier
12. @Retention(RUNTIME)
13.

@interface Dog {}
```

The qualifier of Cat implementation is as follows:

```
static
static
static
static
static
static
10.
11. @Qualifier
12. @Retention(RUNTIME)
13.
@interface Cat {}
```

Note that an annotation that is a qualifier needs to be marked as a qualifier using the annotation

The following is the Dog implementation:

```
1. @Dog
class Dog extends Animal {
3.      ...
4. }
```

The following is the Cat implementation:

```
1. @Cat
class Cat extends Animal {
3.      ...
4. }
```

Now we can inject using the qualifiers, as shown in the following example:

```
@Inject
@Dog
private Animal dog; // Injecting dog

@Inject
@Cat
private Animal cat; // Injecting cat
```

You can define a default implementation. If you define a default implementation in case you inject a bean without using a qualifier, then the implementation selected will be the default. Look at the following example where we define the **Dog** as the default implementation:

```
@Default
public class Dog extends Animal {
...
}
```

```
@Inject
private Animal dog; // Injecting dog
```

Injecting a resource

In our preceding examples, we have injected many CDI beans that are a simple Java instance, but sometimes we need to inject a Jakarta EE resource, that is, an instance that represents a Jakarta EE resource, like JMS queue, data sources, and others. To inject a Jakarta EE resource using the CDI, you should use the which is a CDI annotation that is used to it. Look at the following example:

```
@Resource(lookup="java:MyDatasource")
Datasource datasource;
```

Note that the CDI will look up the data source using the JNDI and will inject it in the data source attribute. The **@Resource** is just used to inject the resources.

Using @PostConstruct

When we instantiate an instance, sometimes we need to initialize some instance's properties and execute some task. Generally, without a CDI, it is done inside a constructor. When we use the CDI, using the constructor to execute the tasks is not recommended, because it is not possible in Java to initialize the fields before the constructor is invoked, and hence CDI cannot inject the dependencies before the constructor is invoked; therefore, they would still be uninitialized (null) inside the constructor block. To solve that, the CDI provides the **@PostConstruct** annotation.

Using you can mark a method to be executed after the constructor runs, that is, after a new instance is created. We can use a **@PostConstruct** in a superclass's method as well, and if this method is not overridden by the subclass, it will be called after the superclass's constructor runs. Note that the methods of superclass will be executed before the methods of the subclass. Look at the following example:

```
1. public class Person {
2.
static Logger LOGGER =
4.
String name;
6.

void
10.     }
```

11.
and setters
13.}

1. public class Developer extends
2.
static Logger LOGGER = Logger.getLogger(Developer.
4.
5.
void
9. }
10.
11.}

As you can see, the class **Developer** extends from **Person** and both have a method marked to be executed after the constructor runs. The **Person.initPerson** will be called and then the

Using @PreDestroy

Similarly, we can want to mark some method to be executed before the bean is destroyed. These methods can be used to prepare the bean instance to be destroyed. An example of this is, in case the bean is using some temp file and this file is not needed after the bean is destroyed, we can mark a method to be executed to remove this file from the file system. Look at the following example that shows how to use the **@PreDestroy**

public void prepareToDestroy(){

```
LOGGER.info("Preparing to destroy processing...");
}
```

We can use **@PreDestroy** both in the subclass and the superclass, and if the method marked to execute the pre-destroy processing is overridden, the superclass's method will not be executed.

Configuring a CDI application

The CDI provides a way to configure the CDI application using a deployment descriptor called beans.xml. This deployment descriptor is optional, and the CDI is able to work without this file. Similar to the other deployment descriptors used in Jakarta EE, the configurations inside the **beans.xml** are used as an additional configuration to the annotation configurations. In case of a conflict of configuration between the annotation and the the **beans.xml** overrides the annotation settings. A CDI application can be either an *implicit bean archive* or *an explicit bean* The following is an explanation of these two CDI application types:

Implicit bean It is an archive (JAR, WAR, EAR, and others) that either does not contain the beans.xml, but contains some beans annotated with a CDI scope, or contains the **beans.xml** with the bean-discovery-mode set to be annotated. In this type, only the annotated beans with a CDI scope will be managed and injected.

Explicit bean It is an archive (.JAR, .WAR, .EAR, and others) containing a **beans.xml** with the bean-discovery-mode attribute set

to all. This file can be with just a root tag In this type, all beans will be managed and injected, even the non-annotated beans.

For a web application the **beans.xml** must be in the **WEB-INF** folder, and to EJB modules or JAR files, it must be in the **META-INF** folder. Look at the following example of **beans.xml** using the explicit bean archive:

```
1. version="1.0" encoding="UTF-8"?>
2.
3.
4.          https://https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"
5.
6.
```

Creating and injecting beans

Firstly, to create and inject beans, and use the CDI features, we'll create a new project that will be used in the entire chapter. Thus, to create a new project called **samplecdi** using the archetype described in *Chapter 1, Introduction to* we will use the following Maven command:

mvn archetype:generate -DarchetypeGroupId=net.rhuanrocha -DarchetypeArtifactId=jakartaee9-war-archetype -DarchetypeVersion=1.0.SNAPSHOT -DgroupId=net.rhuanrocha -DartifactId=samplecdi

Note that you can choose another and/or

As the first example, we'll create a CDI bean called **HelloWorld** with a **RequestScoped** and a CDI bean called **HelloWorldSession** with a and both will have the 'hello world' message within the time of the bean initialization. The **HelloWorld** and **HelloWorldSession** bean will be printed to the browser by the **TestServlet** class according to a request parameter called Look at the following code:

Bean using the

1. *import jakarta.annotation.PostConstruct;*
2. *import jakarta.enterprise.context.RequestScoped;*

```
3. import java.time.LocalDateTime;
4. @RequestScoped
5. public class HelloWorld {
String message;
LocalDateTime date;
void


= "Hello
= LocalDateTime.now();
12.     }
String {
message;
15.     }
void message) {
= message;
18.     }
LocalDateTime {
date;
21.     }
void date) {
= date;
24.     }
25.}
```

Bean using the

```
1. import jakarta.annotation.PostConstruct;
2. import jakarta.enterprise.context.SessionScoped;
3. import java.io.Serializable;
4. import java.time.LocalDateTime;
5.
```

```
7. public class HelloWorldSession implements Serializable {
8.
String message;
10.
LocalDateTime date;
12.


void
= "Hello
= LocalDateTime.now();
17.     }
18.
String {
message;
21.     }
22.
void message) {
= message;
25.     }
26.
LocalDateTime {
date;
29.     }
30. }
```

TestServlet used as an HTTP endpoint:

```
import net.rhuanrocha.samplecdi.beans.HelloWorld;
import net.rhuanrocha.samplecdi.beans.HelloWorldSession;


1. import net.rhuanrocha.samplecdi.beans.HelloWorld;
```

```
2. import net.rhuanrocha.samplecdi.beans.HelloWorldSession;
3.
4. import jakarta.inject.Inject;
5. import jakarta.servlet.ServletException;
6. import jakarta.servlet.annotation.WebServlet;
7. import jakarta.servlet.http.HttpServlet;
8. import jakarta.servlet.http.HttpServletRequest;


9. import jakarta.servlet.http.HttpServletResponse;
10. import java.io.IOException;
11. import java.util.Objects;
12.
=
14. public class TestServlet extends HttpServlet {
15.
HelloWorld helloWorld;
18.
HelloWorldSession helloWorldSession;
21.
void req, HttpServletResponse resp) throws IOException {
23.
type =
25.
== null || {
+ " " + helloWorld.getMessage());
28.             }
+ " " + helloWorldSession.getMessage());
31.             }
32.     }
33. }
```

To test this code, you should open the browser and try to access the following URL:

As we are using the Wildfly, the following is the URL to test:

http://localhost:8080/samplecdi-1.0.SNAPSHOT/test

It will return the **${time} Hello** where the **${time}** is the exact moment of the bean creation. Note that the bean that returned to the browser is **RequestScoped** and the **${time}** should change for each request.

To test the bean with you should use the parameter type. Look at the following example:

http://localhost:8080/samplecdi-1.0.SNAPSHOT/test?type=session

Note that the bean that returned to the browser is **SessionScoped** and the **${time}** should not change for each request.

CDI life cycle

Each CDI bean has a life cycle, and it is defined according to the scope of the bean. To a CDI bean with the scope is based on the HTTP request life cycle. Thus, the bean instance using **RequestScoped** will be alive while the request is alive. After the request is ended, the bean instance using **RequestScoped** will die. The scope already was explained in an earlier topic, but in this topic, we will cover the **ConversationScoped** with more details, because with the the developer can handle a CDI bean's lifecycle programmatically. It is very interesting, but the developer should be careful when using it. Because of that, we'll explain it here, in a separate topic.

Similar to a a **ConversationScoped** can be used to keep the bean state cross request, but differently from using the the scope can be demarcated explicitly by the application. To make the conversation demarcation, you should use the **Conversation** class from CDI. Look at the following example:

```
@Inject Conversation conversation;
```

To make the beginning of the scope, the **begin()** method from **Conversation** should be called and to schedule, the scope to end the **end()** method from **Conversation** should be called. Note that the **end()** method makes a schedule to end. With this, after calling the **end()** method, the bean will not be ended, but it is scheduled to be ended at the end of the request. Look in the

following example, where I have a class called **ShoppingCart** that keeps the number of items in the class:

```
1.  @ConversationScoped

class ShoppingCart implements Serializable {
3.
4.      @Inject
5.      private Conversation conversation;
6.
7.      private Long itemNumber;
8.
9.      @PostConstruct
10.     public void
11.         =
12.     }
13.
14.     public void
15.         ++;
16.     }
17.
18.     public Long
19.         return
20.     }
21.
22.     public void
23.         {
24.             conversation.begin();
25.         }
26.
27.     }
28.
```

```
29.         public  void
30.

31.                     conversation.end();
32.                 }
33.
34.         }
35.
36.         public  String
37.                 return  conversation.getId();
38.         }
39.  }
```

TestConversationScopeServlet class:

```
1.
class  TestConversationScopeServlet  extends  HttpServlet  {
3.
4.          @Inject
5.          private  ShoppingCart  shoppingCart;
6.
7.          private  final  String  templateHtml  =  "%dspan>
8.
9.          public void req, HttpServletResponse resp) throws
ServletException, IOException {
10.
11.                 resp
12.                         .getWriter()
13.                         .println(String.format(templateHtml,
shoppingCart.getItemNumber()));
14.
```

```
15.       }

16.
17.       public void req, HttpServletResponse resp) throws
ServletException, IOException {
18.
19.             String action =
20.
21.
22.                   shoppingCart.startConversation();
23.             }
24.             else
25.                   shoppingCart.endConversation();
26.             }
27.
28.                   shoppingCart.increaseItemNumber();
29.             }
30.
31.             String cid = (shoppingCart.getConversationId() == null) ?
"" : shoppingCart.getConversationId());
32.
33.       }
34.
35. }
```

In the preceding example, we have a servlet responding in We have the **GET** request, which returns the item number of the cart, and the **POST** request that can be used either to increase the item number, start a conversation, or end a conversation. Note that we return the cid header. It is not required, but we are

returning this header because it will help us to use it in the **curl** request. The **cid** is used to help the Jakarta EE context find the conversation used by the user, and it should be sent as a URL parameter. In the following section, we will do the tests using **-X POST http://localhost:8080/samplecdi-1.0-SNAPSHOT/testconversationscope?action=startConversation -v -c cookie.txt**

Note that we used **-c cookie.txt** to save the cookie in It will be used in the next request, to use the same session id. Besides that, the action parameter was sent with value, and it indicates to start the long-running conversation. It returns the following:

```
> POST /samplecdi-1.0-SNAPSHOT/testconversationscope?
action=startConversation HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< cid: 1
< Connection: keep-alive
* Added cookie JSESSIONID="7-
jdBg56l80lNIYC480cpaD_9xN2EYoIWwLmnYjo.localhost" for domain
localhost, path /samplecdi-1.0-SNAPSHOT, expire 0
< Set-Cookie: JSESSIONID=7-
jdBg56l80lNIYC480cpaD_9xN2EYoIWwLmnYjo.localhost;
path=/samplecdi-1.0-SNAPSHOT
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

Note that the CID was returned. Now, we will make a request to increase the item number as follows:

curl -X POST http://localhost:8080/samplecdi-1.0-SNAPSHOT/testconversationscope?cid=1 -v -b cookie.txt

Note that the **-b cookie.txt** is used to read the cookies saved from the previous request. Besides that, the **cid** parameter was used to pass the conversation ID. It returns the following:

```
> POST /samplecdi-1.0-SNAPSHOT/testconversationscope?cid=1 HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Cookie: JSESSIONID=7-jdBg56l80lNIYC480cpaD_9xN2EYoIWwLmnYjo.localhost
>
< HTTP/1.1 200 OK
< cid: 1
< Connection: keep-alive
< Content-Length: 0
* Connection #0 to host localhost left intact
```

Now we can check the item number value making a **GET** request as **http://localhost:8080/samplecdi-1.0-SNAPSHOT/testconversationscope?cid=1 -v -b cookie.txt**

Note that we are using the **cookie.txt** in all the requests. It is because we need to use the same HTTP session for all the requests. It returns the following:

```
GET /samplecdi-1.0-SNAPSHOT/testconversationscope?cid=1
HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080

> Accept: */*
> Cookie: JSESSIONID=7-
jdBg56l80lNIYC480cpaD_9xN2EYoIWwLmnYjo.localhost
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 41
<
1
```

To schedule the bean to be ended, you can apply the following **curl** command:

```
curl -X POST "http://localhost:8080/samplecdi-1.0-
SNAPSHOT/testconversationscope?action=endConversation&cid=1" -b
cookie.txt -v
```

Using producer method, producer field, and dispose method

The example exposed here was about injecting simple Java classes as a CDI bean, but in the real world, we have some cases where we need to inject a complex Java class. In other words, we have a class that should execute a process to create and destroy an instance. As an example, we can create an instance of the class to represent a database connection. With this, this instance needs to execute a process to connect to the database and then create this instance to represent this connection. Similarly, it should end the real connection when the instance is destroyed. One pattern that can be used to do it is the method factory, to create the instance, but how can we do that using the CDI mechanism?

Producer method

The producer method is a CDI feature that permits us to create a method factory to produce the instances of a CDI bean. With this, we create a method that has the building logic to some CDI beans, and this method is executed when a new instance of the respective CDI bean needs to be created. To mark some method as a producer method, we should configure this method with the **@Produces** annotation. When an application has many producer methods, the application should know what the producer method is used to call. To make this differentiation, we create a qualifier. We'll see it in the example covered in the following section. In the preceding example, we will have a class called **MessageWriter**

that will write the messages to file in the file system. This class has a constructor with arguments, so it cannot be injected using CDI without a producer, because a CDI bean should have a constructor without parameters. Look at the following class:

```
class MessageWriter implements Serializable {
2.
3.      private File file;
4.
5.      public path){
6.          file = new File(path);
7.      }
8.
9.      public void message){
10.
11.         outputStream = new {
12.
13.             outputStream.write(message);
14.             outputStream.newLine();
15.
16.         } catch (FileNotFoundException e) {
17.             e.printStackTrace();
18.         } catch (IOException e) {
19.             e.printStackTrace();
20.         }
21.     }
22.
23.     public void
24.         file.delete();
25.     }
26. }
```

Note that this class has a constructor that receives a path to file as an argument and creates the file in the file system. The method **add(String message)** adds a new message to the file. The **clean()** method removes the file from the file system.

Now we'll create the producer method to produce the instance of To create the producer, we'll create a class **MessageWriterProducer** and an annotation called **Message** as annotation:

```
6.
7.  @Qualifier
8.  @Retention(RetentionPolicy.RUNTIME)
9.  1.
10.
@interface  Message  {
12.  }
```

MessageWriterProducer class:

```
net.rhuanrocha.samplecdi.beans.MessageWriter;
net.rhuanrocha.samplecdi.qualifiers.Message;
3.
jakarta.enterprise.inject.Produces;
java.sql.Timestamp;
java.time.LocalDateTime;
java.util.logging.Logger;

8.
class  MessageWriterProducer  {
```

```
10.
11.         private static Logger logger = Logger.
12.         private final String HOME_PATH =
13.         private final String NAME_PATTERN =
14.
15.         @Produces
16.         @Message
17.         public MessageWriter build (){
18.
19.                 String fileName = String.format(NAME_PATTERN,
Timestamp. valueOf(LocalDateTime.now()).getTime());
20.                 file message: + fileName );
21.                 return new MessageWriter(HOME_PATH + fileName);
22.
23.         }
24. }
```

Note the build method generates a name to file and creates the new instance of **MessageWriter** using the path received as an argument and the file name created.

Now we can inject the **MessageWriter** using CDI. To test this injection, we'll create a servlet called **MessageServlet** that will receive a message as the parameter and will write the message to file using the **MessageWriter** injected. Look at the following example:

```
net.rhuanrocha.samplecdi.beans.MessageWriter;
net.rhuanrocha.samplecdi.qualifiers.Message;
3.
```

```
jakarta.inject.Inject;
jakarta.servlet.ServletException;
jakarta.servlet.annotation.WebServlet;
jakarta.servlet.http.HttpServlet;
jakarta.servlet.http.HttpServletRequest;
jakarta.servlet.http.HttpServletResponse;
java.io.IOException;
11. import java.util.Objects;
12.
13.
class MessageServlet extends HttpServlet {
15.
16.
17.
18.         private MessageWriter messageWriter;
19.
20.         public void req, HttpServletResponse resp) throws IOException
{
21.
message =
&& !message.isEmpty()){
24.                     messageWriter.add(message);
25.             }
26.


added:
28.
29.       }
30.}
```

Note that we used the qualifier to indicate the producer method to be executed. The injection point and the producer method

should have the same qualifier.

To test this code, we can send an HTTP request using the **POST** method and send a parameter called a Look at the following example using

curl -X POST "http://localhost:8080/samplecdi-1.0-SNAPSHOT/messages" -v -d "message=hi"

Note that **-d "message= hi"** is used to send the parameter. The following is the return:

```
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> POST /samplecdi-1.0-SNAPSHOT/messages HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 11
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 11 out of 11 bytes
< HTTP/1.1 200 OK
< Connection: keep-alive

< Content-Length: 19
<
Message added: hi
```

Note that a file will be created in

Producer field

Another way to produce the instances of classes using the CDI injection mechanism is by using the producer field.

The producer field is a feature from CDI that permits us to mark some instances of class to be used by the CDI injection. With this, the same instance of a class will be used when the CDI injection tries to inject this class. As an example, we will cover the same scenario covered earlier but using the producer field. To differentiate the qualifier of the preceding example from the next example, we'll create a new qualifier called Look at the following example:

MessageField annotation:

```
1. import
2. import
3. import
4. import
5. import
6.
7. @Qualifier
8. @Retention(RetentionPolicy.RUNTIME)
9.
10. public @interface MessageField {}
```

MessageWriterProducer class:

```
1. import javax.enterprise.inject.Produces;


2. import java.sql.Timestamp;
3. import java.time.LocalDateTime;
4. import java.util.logging.Logger;
5.
6. public class MessageWriterProducer {
7.
static Logger logger = Logger.
final String HOME_PATH =
final String NAME_PATTERN =
11.
14.       MessageWriter messageWriter;
15.
17.              String fileName = String.format(NAME_PATTERN,
Timestamp. valueOf(LocalDateTime.now()).getTime());
file message via producer field: + fileName );
= new MessageWriter(HOME_PATH + fileName);
20.       }
21.
22. }
```

Note that the instance is created inside the constructor and is used every time we need to inject the respective instance. To test this injection, we'll create a servlet called **MessageServlet** that will receive a message as the parameter and will write the message to file using the **MessageWriter** injected. Take a look at the following example:

```
jakarta.inject.Inject;
jakarta.servlet.ServletException;
jakarta.servlet.annotation.WebServlet;
jakarta.servlet.http.HttpServlet;
jakarta.servlet.http.HttpServletRequest;
jakarta.servlet.http.HttpServletResponse;
java.io.IOException;
java.util.Objects;
9.
10.
11. public class MessageServlet extends HttpServlet {
12.
13.
14.
15.         private MessageWriter messageWriter;
16.
17.         public void req, HttpServletResponse resp) throws IOException
{
18.
19.             String message =
20.             && !message.isEmpty()){
21.                 messageWriter.add(message);
22.             }
23.
added:


25.
26.     }
27.}
```

Note that the change in **MessageServlet** is just the **@MessageField** qualifier.

To test this code, we can send an **HTTP** request using the **POST** method and send a parameter called a message. Look at the following example using

curl -X POST "http://localhost:8080/samplecdi-1.0-SNAPSHOT/messages" -v -d "message=hi"

Note that **-d "message= hi"** is used to send the parameter. The following is the return:

```
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> POST /samplecdi-1.0-SNAPSHOT/messages HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 11
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 11 out of 11 bytes
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 19
<
```

Message added: hi

A file will be created in

Dispose method

We have seen how to produce an instance using a CDI producer, but sometimes we want to cleanup this instance or execute some tasks to remove this instance.

The dispose method permits us to mark a method to be executed to remove the instance produced. With this, when the CDI needs to remove this instance, it calls the marked method to execute a task that is needed to remove the instance. To be a dispose method, this method should have one parameter — the instance to be removed — representing the instance produced, and this parameter should be annotated with The dispose method could be just used for instances produced by a producer and it should be the same type of the instance reproduced and the same qualifier of the producer, whether the producer has used it or not. Look at the following example:

```
void @Message MessageWriter messageWriter)          {
2.
3.              file message" );
4.
5.  }
```

Note that the parameter **messageWriter** was annotated with that is, a CDI annotation to dispose methods, and it was annotated

with **@Message** that was the qualifier used by the producer. Look at the complete classes as follows:

```
net.rhuanrocha.samplecdi.beans.MessageWriter;
net.rhuanrocha.samplecdi.qualifiers.Message;


jakarta.enterprise.inject.Disposes;
jakarta.enterprise.inject.Produces;
java.sql.Timestamp;
java.time.LocalDateTime;
java.util.logging.Logger;
8.
class MessageWriterProducer {
10.
11.        private static Logger logger = Logger.
final String HOME_PATH =
final String NAME_PATTERN =
14.
MessageWriter build (){
18.              String fileName = String.format(NAME_PATTERN,
Timestamp. valueOf(LocalDateTime.now()).getTime());
file message: + fileName );
new MessageWriter(HOME_PATH + fileName);
21.       }
22.
void @Message MessageWriter messageWriter) {
file message" );
25.              messageWriter.clean();


26.       }
27.
```

28.}

Creating a CDI interceptor

When we are developing an application, many a times, we need to work with some cross-cutting tasks (such as logging, authorization, or auditing). Working with cross-cutting tasks is not so easy, because it is harder to provide a good separation of responsibility and code reuse. One approach to solve it is **Aspect-Oriented Programming** that allows the separation of the cross-cutting concerns. But how can we use the AOP with CDI and create a cross-cutting task in these cases?

CDI Interceptor is a feature that provides a way to use the AOP based on the interceptors. With this, we can separate the cross-cutting concepts from the business logic and put them inside an interceptor that is configured to intercept some methods or classes and execute its tasks.

A CDI interceptor is a class that wraps the call to a method — this method is called a target method — that runs its logic and proceeds the call either to the next CDI interceptor, if it exists, or the target method. A targeting method could have many CDI interceptors and these interceptors can be configured to run in a specific order. The following are the steps to create a CDI interceptor:

Create an **@InterceptorBinding** to bind the interceptor.

Create a class to be the interceptor.

Annotate the CDI interceptor class with **@Interceptor** and the qualifier created.

Create the method to be executed when the call is intercepted. This method should be annotated with

Declare the interceptor inside the

Configure the target method annotating it with the

These steps will be covered in detail in the following section. The CDI interceptor order can be configured using the **@Priority** annotation or be the order of declaration inside the If the CDI interceptor uses the **@Priority** annotation, this interceptor does not need to be declared inside the

To show you how to use the CDI interceptor, we'll cover an example of two interceptors, one to validate whether a user is authenticated or not, and the other to make the auditing about the access of the functionality. The following are the classes used in this example:

AuthenticationDatasource: The class representing a data source that contains the username and password.

AuthenticationServlet: The class responsible to receive the authentication request and run the authentication logic.

AuthenticationInterceptor: The class responsible to intercept the request to some secured area and validate whether the user can access this area or not.

Authentication: An annotation that is an **@InterceptorBinding** to

AuditoringInterceptor: The class responsible to intercept the requests to some defined areas. This class will just print a log about who is accessing the method and what is the method.

Auditing: An annotation that is an **@InterceptorBinding** to

SecuredServlet: A servlet that receives a request and makes a call to a secured CDI bean. This servlet returns an HTTP status 403 in case the user is not authenticated.

SecuredBean: The CDI bean that is secured. It should be accessed only by the authenticated users.

Firstly, we will create the **AuthenticationDatasource** and **AuthenticationServlet** classes, to help the user be able to authenticate the application. Look at the following example:

class AuthenticationDatasource implements Serializable {

3.

```
4.      private Map userDatasource;
5.
6.      @PostConstruct
7.      public void
8.          userDatasource = new ConcurrentMap<>();
9.
10.
11.     }
12.
13.     public boolean username, String password){
14.
15.         == null || password ==
16.             return
17.         }


18.
19.             return
20.         }
21.
22.         return
23.
24.     }
25. }
```

Note that just as a test, we have initialized the **userDatasource** Map with the hardcoded users, that is, the users that can be authenticated. Note that the **userDatasource** is representing a database, but it persists in memory. Furthermore, this class has the method called validate, which validates a username and password according to Now we can create the Look at the following example:

```
1.
2. public class AuthenticateServlet extends HttpServlet {
3.
4.       @Inject
5.       private AuthenticationDatasource authenticationDatasource;
6.
7.       public void req, HttpServletResponse resp) throws
ServletException, IOException {
8.              String username =
9.              String password =
10.              == null || password ==


11.
resp.sendError(HttpServletResponse.SC_BAD_REQUEST);
12.
13.             }
14.
15.
16.
17.             }
18.
19.                   resp.sendError(HttpServletResponse.SC_FORBIDDEN);
20.              }
21.
22.
23.       }
24. }
```

Note that the servlet receives the username and password and validates these parameters. If it is valid, then the user is logged,

else the servlet returns an HTTP status 403. Okay, so our application is able to do the authentication. Now we want to create the SecuredBean and make the authentication valid to this servlet. To do that, we'll create the Authentication annotation and the **AuthenticationInterceptor** class. Look at the following example:

```
1.  import jakarta.interceptor.InterceptorBinding;
2.  import
3.

ElementType.TYPE})
@interface Authentication {
9.  }
```

Note that it is annotated with We have the **AuthenticationInterceptor** as follows:

```
1.  import net.rhuanrocha.samplecdi.exceptions.
AuthenticationException;
2.
3.  import jakarta.inject.Inject;
4.  import jakarta.interceptor.AroundInvoke;
5.  import jakarta.interceptor.Interceptor;
6.  import jakarta.interceptor.InvocationContext;
7.  import jakarta.servlet.http.HttpServletRequest;
8.  import java.io.Serializable;
9.
12.  public class AuthenticationInterceptor implements Serializable {
13.
HttpServletRequest request;
16.
```

Object invocationContext) throws Exception {

19. String username = (String) request.getSession().
20.
(username ==
new is not
23. }
24.
invocationContext.proceed();
26. }
27. }

In the preceding example, the **AuthenticationInterceptor** is annotated with **@Interceptor** and that is the interceptor binding. The method that will execute in the interception should be annotated with To proceed with the execution to the next interceptor or the target method, we should call the

Now, we should declare this CDI interceptor at the that is, inside the WEB-INF. Look at the following example:

version="1.0" encoding="UTF-8"?>

https://https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"

Now, we can mask some method or class to be intercepted by Based on our scenario, we'll intercept the calls to SecuredBean, then we will create the SecuredBean and annotate it with that is the intercept binding of Look at the following example:

```
@ApplicationScoped

public class SecuredBean {

@Authentication

    public String generateText(String username) throws
AuthenticationException {

return

    }

}
```

Note that the **AuthenticationInterceptor** will intercept the method Now, we should create the **SecuredServlet** that will receive the request and will call the Look at the following code:

```
import net.rhuanrocha.samplecdi.beans.SecuredBean;

import net.rhuanrocha.samplecdi.exceptions.AuthenticationException;

import jakarta.inject.Inject;

import jakarta.servlet.ServletException;

import jakarta.servlet.annotation.WebServlet;

import jakarta.servlet.http.HttpServlet;

import jakarta.servlet.http.HttpServletRequest;


import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;


{
```

```
securedBean;

public void req, HttpServletResponse resp) IOException {

String username =

try {


        } catch e) {


            e.printStackTrace();

        }

    }

}
```

Now we can test it. To access the secured area, apply the following **curl** command:

curl http://localhost:8080/samplecdi-1.0-SNAPSHOT/secured -v -b cookie.txt

It should return an HTTP status 403. Look at the following example of return:

```
> GET /samplecdi-1.0-SNAPSHOT/secured HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>

< HTTP/1.1 403 Forbidden
< Connection: keep-alive
```

Now, to authenticate a user, we can use the following **curl** command:

```
curl -X POST "http://localhost:8080/samplecdi-1.0-SNAPSHOT/authenticate" -v -d "username=rhuan&password=rhuan" -c cookie.txt
```

It should return an HTTP status 200. Look at the following example of return:

```
> POST /samplecdi-1.0-SNAPSHOT/authenticate HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 29
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 29 out of 29 bytes
```

< HTTP/1.1 200 OK
< Connection: keep-alive

Now, we can access the **/secured** again using the following **curl** command:

curl http://localhost:8080/samplecdi-1.0-SNAPSHOT/secured -v -b cookie.txt

Now, it should return an HTTP status 200 instead of 403, as **GET /samplecdi-1.0-SNAPSHOT/secured HTTP/1.1**

> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*

> Cookie: JSESSIONID=22P2PYGflR1PVWUQVbFq-vDxHHA4grrkUBaIbCxo.localhost
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 14

Now, our application is already for the authenticated and validated users. Now we want to add the **AudithoringInterceptor** that will intercept the SecuredBean as well. In our scenario, we want the **AudiroringInterceptor** to execute before the With this, we'll create the Audithoring annotation, which is an interceptor binding, and the Look at the following example:

```
import jakarta.interceptor.InterceptorBinding;

import

@Inherited

@InterceptorBinding

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.METHOD, ElementType.TYPE})

Auditoring {

}
```

Now we will create the as follows:

```
import jakarta.inject.Inject;

import jakarta.interceptor.AroundInvoke;

import jakarta.interceptor.Interceptor;

import jakarta.interceptor.InvocationContext;

import jakarta.servlet.http.HttpServletRequest;
```

```
import java.io.Serializable;


import java.util.logging.Logger;

java.lang.String.*;

@Auditoring

@Interceptor

{

Logger logger =

String PATTERN_AUDIT = "username:%s ; class called:%s ; method

@Inject

private HttpServletRequest request;

@AroundInvoke

public Object invocationContext) throws Exception {

        String username = (String)
```

```
        String className =
invocationContext.getTarget().getClass().getName();

        String methodName =
invocationContext.getMethod().getName();

==

        }

            logger.info(format(PATTERN_AUDIT,username,className,
methodName));

        }

return invocationContext.proceed();

    }

}
```

Now, we should add this interceptor to the **beans.xml** configuration. Note that the order of the execution of interceptors is the order of declaration inside the The following is an example of the **beans.xml** after the update:

version="1.0" encoding="UTF-8"?>

https://https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"

Now, we should annotate the **SecuredBean** with to bind the interceptor to the method. Look at the following example:

@ApplicationScoped

{

@Auditoring

@Authentication

public String username) throws AuthenticationException {

```
    }

}
```

Now, we can retest it by accessing the /secured again, as **http://localhost:8080/samplecdi-1.0-SNAPSHOT/secured -v -b cookie.txt**

Now it should return an HTTP status 200, as **GET /samplecdi-1.0-SNAPSHOT/secured HTTP/1.1**

```
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Cookie: JSESSIONID=22P2PYGflR1PVWUQVbFq-vDxHHA4grrkUBaIbCxo.localhost
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 14
```

Now, if you check the log file or console, you should see a log like it, as follows:

```
INFO [net.rhuanrocha.samplecdi.interceptors.AuditoringInterceptor] (default task-1) username:anonymous ; class called:net.rhuanrocha.samplecdi.beans.SecuredBean$Proxy$_$$_WeldSubclass ; method called:generateText
```

If you want to define a priority without changing the you can use the **@Priority** annotation and pass an integer value to this. Look at the following example:

```
@Interceptor
@Authentication
@Priority(200)

public class AuthenticationInterceptor implements Serializable {
.....
}
```

The interceptor configured with the smaller priority is executed before. If the **@Priority** is defined to the CDI interceptor, it does not need to be declared in the

The CDI interceptor is a good tool to use Aspect-Oriented Programming using the Jakarta EE approach, taking advantage of the Jakarta EE context.

Creating a CDI decorator

When we are developing an application, sometimes it is interesting to add the behavior related to the business logic to some class, but without changing the class. If we analyze, we can do it by using the CDI interceptor, which was covered previously, but it is not the correct approach. The CDI interceptor was created to work with the cross-cutting concepts, and in this case, we want to add the behaviors that are not a cross-cutting concept but a business logic.

The CDI decorator permits us to add new behaviors to some bean without changing the bean but by creating a class that will intercept this bean and perform the auditioning of the behavior.

To show the CDI decorator, we'll create a scenario to calculate the profit of some companies. In this scenario, we'll apply a formula (income - spent). However, before applying the formula, we should discount the tribute from the income. The following are the classes involved in our example:

Calculator: The interface that the CDI bean and the decorator should implement.

The CDI bean that contains the logic to calculate the profit.

TributeDecorator: The CDI decorator used to add the tribute calculus to profit.

ProfitServlet: The servlet used to test the CDI decorator.

As we can see, to work with the CDI decorator, we should create an interface that is implemented both by the CDI bean and the CDI decorator. Furthermore, we should declare the CDI decorator inside the **beans.xml** and define an execution order for them by using the **@Priority** annotation. The following is an example of how to implement this solution:

import java.math.BigDecimal;

{

public BigDecimal income, BigDecimal spent);

}

As you can see, this interface has the method called This method has the logic to apply the calculus and be implemented by the CDI bean and the CDI decorator. The following is the CDI bean called

import jakarta.enterprise.context.ApplicationScoped;

import java.math.BigDecimal;

```
import java.math.RoundingMode;

@ApplicationScoped

{

public BigDecimal income, BigDecimal spent){

return RoundingMode.HALF_UP);

    }

}
```

Note that this bean is In this class, we are returning the profit as a **BigDecimal** with two decimal places. Now we want to add the tribute discount to this calculus. To do that, we have the CDI decorator as follows:

```
import jakarta.decorator.Decorator;

import jakarta.decorator.Delegate;

import jakarta.inject.Inject;



import java.math.BigDecimal;
```

```
import java.math.RoundingMode;

@Decorator



@Inject

@Delegate

private Calculator calculator;

@Override

public BigDecimal income, BigDecimal spent) {

        BigDecimal newIncome = RoundingMode.HALF_UP));

return calculator.calculate(newIncome, spent);

    }

}
```

Now we should register this CDI decorator in our The following is an example of **beans.xml** registering the CDI decorator:

version="1.0" encoding="UTF-8"?>

https://https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd"

Now, we'll create the **ProfitServlet** to test the CDI decorator. The following is the

```
import jakarta.inject.Inject;

import jakarta.servlet.ServletException;

import jakarta.servlet.annotation.WebServlet;

import jakarta.servlet.http.HttpServlet;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
```

```
import java.math.BigDecimal;

import java.math.RoundingMode;



{



calculator;

public void req, HttpServletResponse resp) IOException {

String income =

String spent =

== null || income.isEmpty()

                    || spent == null || spent.isEmpty()){



        }

BigDecimal profit = calculator.calculate(

                    new
```

```
                    new

    }

}
```

As we can see, in the preceding example, we are injecting the **Calculator** interface using CDI and then calling the calculate method, passing the income and spent parameters sent in the HTTP request.

Now, we can test it. The following is a **curl** command that can be used to test it:

```
curl "http://localhost:8080/samplecdi-1.0-SNAPSHOT/profit?
income=1000&spent=200" -v
```

It will generate the following output:

```
> GET /samplecdi-1.0-SNAPSHOT/profit?income=1000&spent=200
HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
```

< Connection: keep-alive
< Content-Length: 15
<
Profit: 700.00

@Decorator

public class TributeDecorator implements Calculator{

...

}

The decorator configured with the smaller priority is executed before.

Using CDI event

We can develop an application in many ways using many approaches and paradigms according to the need. One of these approaches is reactive programming, which permits us to develop the codes that react to a data stream and perform some tasks in an asynchronous way. With this, we generate an event as a data stream and some observers react to these events.

The CDI Event is a feature from CDI that permits us to use the reactive programming approach using CDI. This feature is based on the Observer pattern, that creates an observer to observe some event and react to them. It can work both in a blocking and a non-blocking way. In a blocking way, the event launcher and observer are executed in the same thread, and in the non-blocking, the event launcher and the observer are executed in different threads.

To implement a solution using the CDI event, we should create the bean that represents the data to launch the event and create an observer to show that we'll use a scenario to send an email using the CDI event. Basically, we will have a servlet that will receive the request to send an email, one bean representing the email data, and an observer. The following is the list of the classes used:

The bean that represents the email data.

The observer used to observe the events from the Email bean.

The servlet used to receive the request and launch the event.

The following is the **Email** bean:

import java.util.Optional;

Email {

to;

subject;

message;

getFrom() {

}

{

=

```
    }

getTo() {

return to;

    }

to) {

= to;

    }

public getSubject() {

return Optional.ofNullable(subject);

    }

subject) {

= subject;

    }

public getMessage() {
```

```
return Optional.ofNullable(message);

    }

message) {

= message;

    }


Email String to, String subject, String message){

        Email email = new Email();


        email.setTo(to);

        email.setSubject(subject);

        email.setMessage(message);

return email;

    }
```

```
}
```

Note that the Email has the static method called which is a method factory. The parameters and methods used here are not the focus of this example. You can put any parameter if needed. Now we need to create the observer. The following is the

```
import net.rhuanrocha.samplecdi.beans.Email;

import jakarta.enterprise.event.Observes;

import jakarta.enterprise.event.ObservesAsync;

import java.util.logging.Logger;

{

Logger logger =

Email email){



    }

Email email){



    }
```

```
email){


//Logic to send email.


                    + " sent from email.getFrom()

                    + " to email.getTo());

    }

}
```

To mark a method as an observer, we need to annotate the method's parameter with the **@Observer** or The **@Observer** is used for the blocking execution and the **@ObserverAsync** is used for the non-blocking execution. Note that the parameter from the method should be the bean that is used on the event launcher, in our case, Now we want to create the **EmailServlet** and launch the event. The following is the example:

```
import jakarta.enterprise.event.Event;

import jakarta.inject.Inject;

import jakarta.servlet.ServletException;
```

```
import jakarta.servlet.annotation.WebServlet;

import jakarta.servlet.http.HttpServlet;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;

import java.util.logging.Logger;


EmailServlet extends HttpServlet {

Logger logger = Logger.getLogger(EmailServlet.class.getName());

@Inject


private Event emailEvent;

doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {

=
```

```
String to =

from == null ||

                || to == null ||

            resp.sendError(HttpServletResponse.SC_BAD_REQUEST);


        }

String subject =

String message =

=


to, subject, message));

event

        }


to, subject, message));
```

event

}

}

}

Note that to launch an event, you should inject the **Event** class into the bean that is used as data. Look at the following code snippet:

```
@Inject
private Event emailEvent;
```

To launch the synchronous event, that is, with the blocking execution, you should call the **fire()** method as follows:

```
emailEvent.fire(Email.of(from, to, subject, message));
```

To launch the asynchronous event, that is, with the non-blocking execution, you should call the **fireAsync()** method as follows:

```
emailEvent.fireAsync(Email.of(from, to, subject, message));
```

To test the sync event, you can send a request using the following **curl** command:

curl -X POST "http://localhost:8080/samplecdi-1.0-SNAPSHOT/email" -v -d "from=rhuan@rhuanrocha.net&to=joao@rhuanrocha.net&subject=Test&message=test"

The following is the return:

> POST /samplecdi-1.0-SNAPSHOT/email HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 74
> Content-Type: application/x-www-form-urlencoded
>
*** upload completely sent off: 74 out of 74 bytes**
< HTTP/1.1 200 OK

This is the log printed. Note that the order and the thread of the log should always be the same to both the lines as follows:

00:08:06,220 INFO [net.rhuanrocha.samplecdi.event.EmailObserver] (default task-1) Email Test sent from rhuan@rhuanrocha.net to joao@rhuanrocha.net
00:08:06,221 INFO [net.rhuanrocha.samplecdi.servlet.EmailServlet] (default task-1) Sync event sent

Now, to test the **async** event, you can use the following **curl** command:

curl -X POST "http://localhost:8080/samplecdi-1.0-SNAPSHOT/email" -v -d "async=true&from=rhuan@rhuanrocha.net&to=joao@rhuanrocha.net&subject=Test&message=test"

The following is the return:

> POST /samplecdi-1.0-SNAPSHOT/email HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 85
> Content-Type: application/x-www-form-urlencoded
>
*** upload completely sent off: 85 out of 85 bytes**
< HTTP/1.1 200 OK

This is the log printed. Note that the order is not guaranteed, and the thread of the log should always be different to both the lines, as follows:

00:12:05,565 INFO [net.rhuanrocha.samplecdi.event.EmailObserver] (Weld Thread Pool -- 4) Email Test sent from rhuan@rhuanrocha.net to joao@rhuanrocha.net
00:12:05,568 INFO [net.rhuanrocha.samplecdi.servlet.EmailServlet] (default task-1) Async event sent

Conclusion

The CDI specification is a very important specification from Jakarta EE, both to the Jakarta EE ecosystem and to the many frameworks, Java tools, and Java projects that use it to build its solutions. Some of these are Quarkus and Microprofile that are not in the scope of this book, but we are quoting them just as an example.

The CDI provides the injection mechanisms, integration with the other specifications, tools to implement the interceptors, decorators, reactive codes, producers, and more, knowing that CDI is very important to any Java developer.

Now, you will be able to use all these CDI features and construct a solution using the CDI in a better way. The project with all the examples shown here is in

In the next chapter, we'll talk about Jakarta RESTful Web Service specification and how we can use it to create the RESTful endpoints.

CHAPTER 4

Jakarta RESTful Web Service

Introduction

In this chapter, we will explain what Jakarta RESTful web service is, how can we create a RESTful resource class, how can we extract request parameters, how can we create a Jakarta RESTful Client, how can we make an asynchronous invocation in the client, and how can we use a **Server-Sent Event** After studying this chapter, you will be able to use the Jakarta RESTFul web service to work with REST and RESTful in your project.

Structure

In this chapter, we will discuss the following topics:

Jakarta RESTful web service

Creating a RESTful resource class

Extracting request parameters

Creating a Jakarta RESTful client

Asynchronous invocation in the client

Using server-sent events

Objectives

The main objectives of this chapter are to enable the readers to learn how to use a Jakarta RESTful Web Service, create RESTful resource class, extract request parameters, validate RESTful resource with Bean Validation, and create a RESTful client. The chapter will also enable them to learn how to use asynchronous invocation in the client and Server-Sent Events efficiently.

Jakarta RESTful web service

To understand what the Jakarta RESTful web service is, we need to understand what the **Representational State Transfer** is, and what is the problem it resolves. Nowadays, integration between applications have become more of a necessity. With this, the manner in which these applications integrate has also evolved a lot. These evolutions brought us the **Application Programming Interface** across the internet that permits us to expose functionalities to be accessed in a decoupled and easier way. In this chapter, we'll not explain in depth the API concepts nor the REST concept, because it is a subject that can generate another book, but we will explain how to implement it using the Jakarta EE approach. However, we need at least a brief introduction about REST. So, let go.REST is an architectural style that is very used to building API to make the communication between applications and/or between the backend and the frontend. This architecture provides a loosely coupled, lightweight, and stateless communication. It is provided over the HTTP protocol that is a stateless protocol, which is ideal to the REST style. In the REST, data and functionalities are exposed as a resource that can be accessed via **Uniform Resource Identifiers** These resources have a contract that might be satisfied to work correctly. A contract means a well-known pattern to consume the resource, which is known by the consumer (client). As it is very lightweight, the contract, in general, is very simple and is around the data and URI, and does not have another complexity as the old remote EJB or SOAP.

Each functionality is a resource, and each resource has a URI. A resource is self-descriptive and can be exposed in many formats (JSON, XML and others). Furthermore, it has a uniform interface that can expose the create, read, update, and delete operations, and provide a stateless communication.

In the frontend and backend communication, when we use REST to provide the communication, it turns the backend and frontend loosely coupled. When used to integrate applications, it turns these applications loosely coupled as well. It is very common to use REST in both the cases.

Okay, we understood that the functionalities are resources and resources are accessed by a URI, providing a loosely coupled with who is accessing it, but we are yet to understand how the REST approach is implemented by Jakarta RESTful Web Service, and how can we use this interesting style using the Jakarta EE approach?

RESTful resource

In the Jakarta RESTful Web Service, the RESTful resource is represented by a class that is annotated with **@Path({some-uri})** and it should be at least one method configured to respond to some HTTP method. This configuration is done via annotations, and the following are the options to configure:

The annotation that configures the method to an HTTP GET request.

The annotation that configures the method to an HTTP POST request.

The annotation that configures the method to an HTTP PUT request.

The annotation that configures the method to an HTTP DELETE request.

The annotation that configures the method to an HTTP HEAD request.

The annotation that configures the method to an HTTP OPTIONS request.

The annotation that configures the method to an HTTP PATH request.

The following is an example of a RESTful resource with the method to respond to an HTTP GET request:

Public

...

@GET

public Response

//Logic

...

}

}

Note that we defined the URI to this resource. Thus, if an HTTP GET request is made to this URI, then the method **list()** will be executed. We will see it in more detail in the following sections.

Besides that, the Jakarta RESTful Web Service provides mechanisms to extract the query parameters and path parameters, as well as define what the format is to consume and produce. These are configured using the annotation as well and they are as follows:

An annotation used to extract the query parameters.

An annotation used to extract the path parameters.

An annotation used to configure the format to consume.

An annotation used to configure the format to the response.

We will see these annotations in more detail in the following sections.

Configuring RESTful applications

To work with Jakarta RESTful web service, you should configure the base URI used by the RESTful resources. This configuration can be done in two ways, which are, either by creating a subclass of **javax.ws.rs.core.Application** (subclass based configuration) or by configuring it in the **web.xml** based configuration).

Subclass based configuration

This approach provides more control to the developer because it permits us to override the way to finding the RESTful resource on the project. But what does *the way to finding the RESTful* means? We'll see it in the following section.

To configure it using the subclass based configuration, we should create a subclass that implements **javax.ws.rs.core.Application** and is annotated with as follows:

{

}

As you can see, the base URI is defined in the In this case, the URL to RESTful resource will be We will see it in more detail when we create a RESTful resource on practices in the following section.

As a default, all classes annotated with that is, all RESTful resources inside the project will be processed, but as I said earlier, using the subclass based configuration, we can override the way to finding the RESTful resource. It can be done overriding the **getClasses** from Look at the following example:

```
{

@Override

public Set> getClasses() {

        Set> classes = new HashSet<>();


return classes;

    }

}
```

As you can see, the **getClasses** method returns a **Set>** containing the resource to be processed. Only the resource inside the set will be processed.

Web.xml based configuration

This approach provides a way to configure the Jakarta RESTful web service without needing to create a subclass and just putting this configuration to In this approach, all the RESTful resources inside the project will be processed. The following is an example of the **web.xml** configuration of the base URI to

Besides that, the **web.xml** based configuration permits us to override the configurations done using the subclass based configuration. Look at the following example:

Note that it is overriding the base URI from defined by the **RESTApplication** subclass to

Creating a RESTful resource class

To create an example of a RESTful resource, first we'll create a new project called **samplerestful** using the archetype described in *Chapter 1, Introduction to* Thus, to create a project, you can use the Maven command as follows:

mvn archetype:generate -DarchetypeGroupId=net.rhuanrocha -DarchetypeArtifactId=jakartaee8-war-archetype -DarchetypeVersion=1.0.SNAPSHOT -DgroupId=net.rhuanrocha -DartifactId=samplerestful

Note that you can choose another **-DgroupId** and/or

After creating a new project, we will configure the Jakarta RESTful using the subclass-based configuration. To do that, we will create the **RESTApplication** that extends as follows:

public class RESTApplication extends Application {

}

Note that the **@ApplicationPath** annotation is configuring the RESTful resource to respond to the requests with base URI Thus, a RESTful resource will respond in

Then, we will create a RESTful resource to list all the data of the users. As we don't have any data yet, the resource will return an empty array. The following is an example of the resource called **UserResource** that has a method to respond to the HTTP GET requests:

```
public class UserResource {

@GET

    @Produces(MediaType.APPLICATION_JSON)

    public Response list(){
```

```
    }

}
```

Note: The @Path annotation is configuring the URI to this resource. Thus, this resource can be accessed by We are testing the RESTful resource deployed in the Wildfly using CURL, as follows:

The @Path annotation is configuring the URI to this resource. Thus, this resource can be accessed by Below we are testing the RESTful resource deployed in the Wildfly using CURL.

curl "http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/users" -v

The output is as follows:

```
* About to connect() to localhost port 8080 (#0)
*   Trying 127.0.0.1...

* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /samplerestful-1.0-SNAPSHOT/resources/users HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/json
```

< Content-Length: 2

<

*** Connection #0 to host localhost left intact**

[]

The resource method can return a **Response** instance or a simple Java object instance that is translated to JSON, XML, or TEXT according to the configuration of resource and the type of object. Look at the following example, where a list of objects of the type **User** is returned:

```
@GET

public List

        List users = new ArrayList();

// Logic

return users;

    }
```

The difference between returning a simple Java object and returning the **Response** is that the **Response** permits us to modify the return, adding the headers and other information to the

return. Particularly, I prefer using the **Response** as it provides more options for managing the response.

Extracting request parameters

When we send a request to some resources, sometimes we need to send some parameters to drive the resource. An example of this is the resource **UserResource** created earlier. Imagine we want to create a **POST** method to create a new user. The **POST** method should receive the user's data to be able to create them. These parameters can be sent in many ways that the Jakarta RESTful is able to work. These parameters can be sent as follows:

URI path (or path Using @PathParam annotation.

Using **@QueryParam** annotation.

Using **@HeaderParam** annotation.

Using **@FormParam** annotation.

Using **@CookieParam** annotation.

Using **@MatrixParam** annotation.

Thus, we will increase the **UserResource** resource, adding a new method called to save, that will be responsible to receive the parameters, and create and save a new user. In our example, the user will be saved in memory, as we are not working with the

database yet. In this book, we have a chapter that will show how to work with a relational database using Jakarta Persistence. As a first example, the parameter will be sent using the To expose the example, we'll create a class to represent the datasource and another to represent the user. The following is the list of classes used:

A Java Bean that represents the user information.

A class that represents the datasource to save the user information. This class saves the information in memory, inside a

It is the class already created in the example earlier, but a new **POST** method will be added in this. This class represents the RESTful resource.

We have the implementation of the example, in the following section.First, we'll create the **User** class that will have the id, name, and email attributes, shown as follows:

import java.util.Objects;

User {

id;

name;

```
email;

getId() {

return id;

    }

id) {

= id;

    }

getName() {

return name;

    }

name) {

= name;

    }

getEmail() {
```

```
return email;

    }

email) {

= email;

    }

@Override

o) {

if == o)

if (o == null || getClass() != o.getClass())

        User user = (User) o;

return Objects.equals(id, user.id);

    }

@Override

public int hashCode() {
```

```
return Objects.hash(id);

    }

User name, String email){

        User user = new User();

        user.setName(name);

        user.setEmail(email);

return user;

    }

}
```

Then we'll create the **UserDatasource** class with the persist, and **ListAll** methods as follows:

```
import

import jakarta.enterprise.context.ApplicationScoped;

import java.util.HashMap;
```

```
import  java.util.List;



import  java.util.Map;

import  java.util.UUID;

import  java.util.stream.Collectors;

@ApplicationScoped

{

private  MapUser>  users;

@PostConstruct

public  void

            users  =  new  HashMap<>();

      }

public  Optional  id){

return  Optional.ofNullable(users.get(id));

      }
```

```
public User persist(User user){

==

                String id = UUID.randomUUID().toString();

                user.setId(id);

        }

        users.put(user.getId(),user);

return user;

    }

public List listAll(){

return users

                    .values()

                    .stream()

                    .collect(Collectors.toList());

    }
```

```
public List listByEmail(String email){


return users

                    .values()

                    .stream()

                    .filter(user -> user.getEmail().equals(email))

                    .collect(Collectors.toList());

    }

}
```

Note that in the persist method, if the user's id is null, then the id is generated using the UUID.

Now, finally, we'll add the **POST** method called save into **UserResource** class which will be responsible to treat all the **POST** requests to the as follows:

```
import jakarta.inject.Inject;

import jakarta.ws.rs.*;
```

```
import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

import java.net.URI;



{

@Inject

private UserDatasource userDatasource;

@GET

@Produces(MediaType.APPLICATION_JSON)

public Response list(){

return Response.ok(userDatasource.listAll()).build();

    }

@POST
```

```
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)

public Response name, String email){

        User user = userDatasource.persist(User.of(name,email));

//returning a HTTP 201 with locator = /users/{id}

return Response

user.getId()))))

                .build();

    }

}
```

Note that to configure the save method to respond to a **POST** request, we should annotate it with The save method has two parameters, i.e., name and email and both are annotated with **@FormParam** passing the name of the form parameter. Look at the following code snippet:

```
public Response save(@FormParam("name")String name,
@FormParam("email") String email)
```

So, the code calls the **UserDatasource.persist** method to persist and return an HTTP code 201 with the URI to the new element created. Note that the **Consumes** should be configured to

Look at the following code snippet:

```
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
```

To test the new method, we can use this curl command to make a **POST** request as follows:

```
curl -X POST "http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/users" -H "Content-Type: application/x-www-form-urlencoded" -d "name=Rhuan&email=rhuan@test.com" -v
```

The following is the return of the **curl** command:

```
> POST /samplerestful-1.0-SNAPSHOT/resources/users HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/x-www-form-urlencoded
> Content-Length: 31
>
* upload completely sent off: 31 out of 31 bytes
< HTTP/1.1 201 Created
< Connection: keep-alive
< Location: http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/users/26bdeda1-e73d-4d88-9c19-7f560bb7bb8d
```

< Content-Length: 0

Note the location header on the response. The following is the URL to the resource created as configured in the **Response** object in our code: **http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/users/26bdeda1-e73d-4d88-9c19-7f560bb7bb8d**

The URL returned at the location will not work yet, as we didn't create a method to respond to this. But throughout this chapter, we'll create this method as a best practice.

Now we can list the user again using the **curl** command to check if the user was saved. To do that, apply the **curl** command as follows:

curl "http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/users" -v

The following is the return of this command:

> GET /samplerestful-1.0-SNAPSHOT/resources/users HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/json
< Content-Length: 87

*** Connection #0 to host localhost left intact**
[{"email":"rhuan@test.com","id":"26bdeda1-e73d-4d88-9c19-7f560bb7bb8d","name":"Rhuan"}]

Note that the ID is different to each user created.

Now, we'll use the URI parameter (or path parameter) to create a method to return a user by ID. With this, the ID will be part of the URL and the resource will identify the user and return if found. In case the user is not found, an HTTP 404 will be returned.

To do that, we'll put the following method into **UserResource** class:

```
@GET


@Produces(MediaType.APPLICATION_JSON)


public Response String id){

        Optional = userDatasource.findById(id);


return Response.status(Response.Status.NOT_FOUND).build();
```

```
        }

return  Response.ok(user.get()).build();

    }
```

Note that we should configure the method with the **@Path** and define the key in this example) that should match the key inside the With this, the value coming in the URL will populate the parameter that matches the key used. The following is the class **UserResource** after adding the **findById** method:

```
import  jakarta.inject.Inject;

import  javax.ws.rs.GET;

import  javax.ws.rs.Path;

import  javax.ws.rs.Produces;

import  javax.ws.rs.QueryParam;

import  javax.ws.rs.Consumes;

import  javax.ws.rs.FormParam;

import  javax.ws.rs.POST;
```

```
import javax.ws.rs.PathParam;

import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

import java.net.URI;



{

@Inject

private UserDatasource userDatasource;

@GET

@Produces(MediaType.APPLICATION_JSON)

public Response list(){

return Response.ok(userDatasource.listAll()).build();

    }

@POST
```

```
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)

public Response name, String email){

        User user = userDatasource.persist(User.of(name,email));

//returning a HTTP 201 with locator = /users/{id}

return Response

user.getId()))))

                    .build();

    }

@GET


@Produces(MediaType.APPLICATION_JSON)

public Response String id){

        User user = userDatasource.findById(id);

==
```

```
return  Response.status(Response.Status.NOT_FOUND).build();

            }

return  Response.ok(user).build();

      }

}
```

To test the new method, you should send a new **POST** request, as it is persisting the data in the memory and all the data is cleaned after the server's restart, as follows:

```
curl -X POST "http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/users" -H "Content-Type: application/x-www-form-urlencoded" -d "name=Rhuan&email=rhuan@test.com" -v
```

Then send a request using the ID in the URL, as follows:

```
curl http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/users/a3666f8e-9aad-4d5f-ac3e-daa29d60e3e3 -v
```

The following will be the return of the **HTTP/1.1 200 OK**

< Connection: keep-alive
< Content-Type: application/json
< Content-Length: 85
*** Connection #0 to host localhost left intact**
{"email":"rhuan@test.com","id":"a3666f8e-9aad-4d5f-ac3e-daa29d60e3e3","name":"Rhuan"}

Note that the ID will be different in each test.

Now, we'll use the **@QueryParam** to extract the query parameter from the URL. In our scenario, we will change the **GET** method to turn it to be able to list the users by email as well. Thus, if we send the email parameter as a query parameter, the backend will return the results based on the parameter if the parameter is not sent; thus, the backend returns all the uses as a list.

With this, we should change the list method inside the **UserResource** class. Look at the following code snippet:

@GET

public Response String email){

!= null){

```
        }


    }
```

As you can see, we included a **String** parameter called email and it is annotated with the It means if a query parameter called email is sent, the email attribute will be populated with the value of the parameter. The following is the **UserResource** class after changing the list method:

```
import jakarta.inject.Inject;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.QueryParam;

import javax.ws.rs.Consumes;

import javax.ws.rs.FormParam;

import javax.ws.rs.POST;
```

```
import javax.ws.rs.PathParam;

import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

import java.net.URI;



{

@Inject

private UserDatasource userDatasource;

@GET

@Produces(MediaType.APPLICATION_JSON)

public Response String email){

!=

return Response

                        .ok(userDatasource.listByEmail(email))
```

```
                .build();

        }

return Response.ok(userDatasource.listAll()).build();

    }

@POST

@Consumes(MediaType.APPLICATION_FORM_URLENCODED)

public Response name, String email){

        User user = userDatasource.persist(User.of(name,email));

//returning a HTTP 201 with locator = /users/{id}

return Response



user.getId())))

                .build();

    }

@GET
```

```
@Produces(MediaType.APPLICATION_JSON)

public Response String id){

        User user = userDatasource.findById(id);

==

return Response.status(Response.Status.NOT_FOUND).build();

        }

return Response.ok(user).build();

    }

}
```

To test the new method, you should send a new **POST** request, as it is persisting the data in the memory and all the data is cleaned after the server's restart, as follows:

```
curl -X POST "http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/users" -H "Content-Type: application/x-www-form-urlencoded" -d "name=Rhuan&email=rhuan@test.com" -v
```

Then send a request with the query parameter in the URL, as follows:

curl http://localhost:8080/samplerestful-1.0-SNAPSHOT/resources/user?email=rhuan@test.com -v

The following is the return of the request:

< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/json
< Content-Length: 87
< Date: Sun, 05 Jul 2020 00:24:03 GMT
<
*** Connection #0 to host localhost left intact**
[{"email":"rhuan@test.com","id":"a5c9b26b-7c28-4242-86c7-3fffb9f22296","name":"Rhuan"}]

If you send a request with the email query parameter that does not match some user, the backend returns an empty list. Note that I don't return a 404 (not found) nor 204 (no content) as the 404 is used in case a resource is not found and the resource in this case is a list — the list exists, but it is empty. The 204 is used to indicate that some operation was executed, and no data needs to be returned, but in our case, we return an empty list. Thus, the more appropriate code to return is 200.

The query parameter has a way to define a default value in case the query parameter is not informed. To do that, we can use the

@DefaultValue annotation, as shown in the following example:

```
@GET

    public Response String email){

!= null){

        }

    }
```

We can use the **@HeaderParam** to extract the parameters from the HTTP header. The following is an example of how to use the

```
@GET

    public Response String email){

!= null){
```

}

}

We can use the **@CookieParam** to extract the parameters from the HTTP cookie parameters. The following is an example of how to use the

@GET

public Response String email){

!= null){

}

}

We can use the matrix parameter as well. It has this pattern, Here, we are not discussing when it is better to be used, but how we can use it with the Jakarta RESTful Web Service. To use it, we can use the as shown in the following example:

@GET

public Response String email){

!= null){

}

}

The **@MatrixParam** is not so used in a common scenario and the most commonly used are the and the

The default value can be applied to these parameter types as well. These annotations make the extracting of the parameters very easy

for the developer. You can implement the RESTful style using the Jakarta Servlet but it will spend a lot of line code. With the Jakarta RESTful Web Service, it is very easy to be done, as we saw in the preceding examples.

Creating a Jakarta RESTful client

Earlier we explained how to expose a RESTful resource using the Jakarta EE approach, but now we want to know how we can consume a resource or API inside our application. One very common characteristic of almost all web applications is needing some kind of integration. The applications need to integrate with the other applications to perform their tasks. In the past, these integrations were done in many ways like CORBA, remote EJB, and SOAP. These integrations don't attempt the contemporaneous necessities, as these are very coupled approaches, mainly the CORBA and the remote EJB. Thus, the approach closer to the contemporaneous necessities is the APIs using RESTful styles as shown in the examples earlier. However, how does the Jakarta EE provide a way to consume it? How does it provide a way to integrate a Jakarta EE application to another application?

The Jakarta RESTful client is a feature from the Jakarta RESTful Web Service that permits us to consume an API in an easier way. To expose an example of how to use the Jakarta RESTful client, we'll create a scenario where we have an application that reads the user from the first one created by us in the example earlier, and then sends an email to this user. With this, we will create a new project with the following maven command:

mvn archetype:generate -DarchetypeGroupId=net.rhuanrocha -DarchetypeArtifactId=jakartaee9-war-archetype -

DarchetypeVersion=1.0.SNAPSHOT -DgroupId=net.rhuanrocha -DartifactId=samplerestfulclient

Thus, we'll create these classes inside the application as follows:

A class that is a Jakarta RESTful resource to receive the email requests.

A class that has the logic to send an email. As it is just an example, this class will not send an email in fact, but log it in the log file to emulate that.

A class that has the logic to integrate to the other application to read the user information. This class has the example of the Jakarta RESTful client.

Thus, we have the **EmailResource** and the **EmailService** as follows; it's not such an important class in our example, so we'll not take the time to explain class:

```
import jakarta.inject.Inject;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;
```

```
import javax.ws.rs.QueryParam;

import javax.ws.rs.Consumes;

import javax.ws.rs.FormParam;

import javax.ws.rs.POST;

import javax.ws.rs.PathParam;

import jakarta.ws.rs.core.Response;


{

@Inject

private UserService userService;

@Inject

private EmailService emailService;

@POST
```

```
public Response String idUser,

@DefaultValue String subject,

String message){

        UserDto user = userService.findById(idUser);

==

return Response.status(Response.Status.NOT_FOUND).build();

        }


return Response.ok().build();

    }

}
```

EmailService class:

```
import java.util.logging.Logger;

EmailService {

Logger logger = Logger.getLogger(EmailService.class.getName());
```

```
String to, String subject, String message){

//Logic to send email.

from %s to %s with a subject %s and message




    }

}
```

The following is the **UserService** class that has the example of the Jakarta RESTful client:

```
import jakarta.ws.rs.client.Client;

import jakarta.ws.rs.client.ClientBuilder;

import jakarta.ws.rs.core.MediaType;

{


id){
```

Client client =

UserDto user = client

}

}

To test it, you can use the following **curl** command:

curl -X POST "http://localhost:8080/samplerestfulclient-1.0-SNAPSHOT/resources/emails/8e75c683-ca20-4a0f-8dc3-de67f3898a5e" -H "Content-Type: application/x-www-form-urlencoded" -d "subject=test&message=test" -v

The following is the return:

> POST /samplerestfulclient-1.0-SNAPSHOT/resources/emails/8e75c683-ca20-4a0f-8dc3-de67f3898a5e HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*

> Content-Type: application/x-www-form-urlencoded
> Content-Length: 25

>
*** upload completely sent off: 25 out of 25 bytes**
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 0

You should put the correct user id in the URL.

Note that the **.get(UserDto.class)** returns a **UserDto** object as the request receives a JSON as a response. In case of no user returned, the **.get(UserDto.class)** returns null. Furthermore, the **.get** method sends an **HTTP GET** request and if you want to send an **HTTP** for example, you can use the The **client.target(...)** returns an instance of **Builder** and the **Builder** has the following methods according to the **HTTP** method:

GET: **Builder.get()**

POST: **Builder.post()**

HEAD: **Builder.head()**

PUT: **Builder.put()**

DELETE: **Builder.delete()**

OPTIONS: **Builder.options()**

TRACE: **Builder.trace()**

PATCH: **Builder.patch()**

We'll see the example of using the **HTTP POST** soon in the following sections. We'll not show each method, but all these methods follow the same logic.

In the example earlier, we got the **UserDto** that was parsed from a JSON return, but it does not give us the HTTP status returned, nor the HTTP header returned, or any other information. To extract these information, we can use a **.get()** method that returns the **Response** object.

Now, we want to receive the user's information and save it by calling the User's resource on the other application instead of finding it there, and then send an email. In this example, we do not care about the scenario and it is a fictitious scenario, and therefore, it is just to show you how to send a **HTTP POST** request using the Jakarta RESTful client. To do this, first we will create a new method in the **EmailResource** class to treat the request. Look at the following method:

```
@POST

public Response name,

String email, @DefaultValue String subject, String message){
```

```
            UserDto user = UserDto.of(name, email);



return Response.ok().build();

            }

return Response.serverError().build();

    }
```

Now we'll add a new method in the **UserService** class. Look at the following method:

```
userDto){



        Form form = new Form();

userDto.getName());

userDto.getEmail());

        Client client = ClientBuilder.newClient();

        Response response = client
```

```
                .post(Entity.form(form));

return response.getStatus() == If status is 201 then it returns true.

    }
```

Note that we create an instance of **Form** and use it in the post method. As the other service returns a HTTP STATUS 201 in case of success, we'll validate it to check if the user was saved. The following is the entire **UserService** class after adding the method:

```
import jakarta.ws.rs.client.Client;

import jakarta.ws.rs.client.ClientBuilder;

import jakarta.ws.rs.client.Entity;

import jakarta.ws.rs.core.Form;

import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

{
```

id){

Client client =

UserDto userDto = client

}

Form form = new

userDto.getName());

userDto.getEmail());

Client client =

Response response = client

return response.getStatus() == If status is 201 then it returns true.

```
    }

}
```

To test it, you can use the following **curl** command:

```
curl -X POST "http://localhost:8080/samplerestfulclient-1.0-SNAPSHOT/resources/emails" -H "Content-Type: application/x-www-form-urlencoded" -d "name=pedro&email=pedro@test.com&subject=test&message=test" -v
```

The following is the return:

```
> POST /samplerestfulclient-1.0-SNAPSHOT/resources/emails HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/x-www-form-urlencoded

> Content-Length: 57
>
* upload completely sent off: 57 out of 57 bytes
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 0
```

Asynchronous invocation in the client

In the real scenario, many a times, the developer has problems and challenges about performance, and needs to figure out some solutions or ways to solve or soften it. One way to soften it is working with the non-blocking processes, calling some process, or resource asynchronously. With an asynchronous call, the resource requested returns the controls at the same time, even if the resource is executing the reprocess. With this, the current process call continues your work even if the requested resource did not finish the work yet.

It's a very interesting feature and we already saw how to implement it using some features from Jakarta EE, but how can we use it with Jakarta RESTful web service to call a service asynchronously?

The Jakarta RESTful client permits us to create an asynchronous call and returns a **Future** or **CompletionStage** object that can be used to retrieve the results and status of the call. As an example, we'll add a new method called **saveAsync()** to that will do the same as the **save()** method, but the **saveAsync()** will do it in an asynchronous way. Look at the following method:

```
userDto, String subject, String message){

        Form form = new Form();
```

userDto.getName());

userDto.getEmail());

Client client = ClientBuilder.newClient();

Future future = client

.async()

.post(Entity.form(form), new InvocationCallback(){

@Override

response) {

== Response.Status.CREATED.getStatusCode()) {

}

User had some issue to

```
                    }

                }

@Override

throwable) {

User had some issue to

                    }

                });

    }
```

As you can see, we need to call the **async()** method, and later, call the **post()** method, passing an instance of **InvocationCallback** that has the **completed()** method, which is called when the execution is completed without error. In case of an error, the **failed()** method is called. Note that the **failed()** method is executed only if the error happened when it tried to connect to the service, but for some reason it was not possible. If the connection was done correctly and the service has any issue, it will return a response with the HTTP status code in the **completed()** method. We have the complete **UserService** as follows:

```
import jakarta.inject.Inject;

import jakarta.ws.rs.client.Client;

import jakarta.ws.rs.client.ClientBuilder;

import jakarta.ws.rs.client.Entity;

import jakarta.ws.rs.client.InvocationCallback;

import jakarta.ws.rs.core.Form;

import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

import java.util.concurrent.Future;

import java.util.logging.Logger;

{

String TARGET_SERVICE=

private Logger logger =

@Inject
```

```
private EmailService emailService;

public UserDto id){

        Client client = ClientBuilder.newClient();

        UserDto userDto = client



return userDto;

    }

userDto){

        Form form = new Form();


userDto.getName());

userDto.getEmail());

        Client client = ClientBuilder.newClient();

        Response response = client
```

```
                .post(Entity.form(form));

return response.getStatus() == If status is 201 then it returns true.

    }

userDto, String subject, String message){

        Form form = new Form();

userDto.getName());

userDto.getEmail());

        Client client = ClientBuilder.newClient();

return client


                .async()

                .post(Entity.form(form), new InvocationCallback(){

@Override

response) {
```

== Response.Status.CREATED.getStatusCode()) {

}

User had some issue to

}

}

@Override

throwable) {

User had some issue to

}

});

}

```
}
```

We should update the **EmailResource** to call the **saveAsync()** method from **UserService** as well. Look at the following method **sendAndSaveUser()** after updating:

```
@POST

public Response name,

String email,

@DefaultValue String subject,

String message){

        UserDto user = UserDto.of(name, email);

        userService.saveAsync(user,subject,message);


return Response.ok().build();

    }
```

To test it, you can use the following **curl** command:

curl -X POST "http://localhost:8080/samplerestfulclient-1.0-SNAPSHOT/resources/emails" -H "Content-Type: application/x-www-form-urlencoded" -d "name=pedro&email=pedro@test.com&subject=test&message=test" -v

The following is the return:

```
> POST /samplerestfulclient-1.0-SNAPSHOT/resources/emails HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/x-www-form-urlencoded
> Content-Length: 57
>
* upload completely sent off: 57 out of 57 bytes
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 0
```

It can be applied to GET or another HTTP method in the Jakarta RESTful client.

Using CompletionStage

As described earlier, we can use **CompletionStage** as well. It is a newer approach than the use of future and provides more features to manage the results. Look at the following example on how it can be implemented using the

```
saveAsync(UserDto userDto, String subject, String message){

        Form form = new Form();

userDto.getName());

userDto.getEmail());

        Client client = ClientBuilder.newClient();

        CompletionStage completionStage = client


                .rx()

                .post(Entity.form(form));

        completionStage.thenAccept(response -> {
```

```
== Response.Status.CREATED.getStatusCode()) {



            }


User had some issue to

            }

        });

    }
```

Note that instead of calling the **async()** method, we are calling the **rx()** method, and then subscribing a function to the **Accept()** method to be executed when the resource request returns some response.

To test it, you can use the following **curl** command:

curl -X POST "http://localhost:8080/samplerestfulclient-1.0-SNAPSHOT/resources/emails" -H "Content-Type: application/x-www-form-urlencoded" -d

"name=pedro&email=pedro@test.com&subject=test&message=test" -v

The following is the return:

```
> POST /samplerestfulclient-1.0-SNAPSHOT/resources/emails HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/x-www-form-urlencoded
> Content-Length: 57
>
* upload completely sent off: 57 out of 57 bytes
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 0
```

Understanding how to use server-sent events

The HTTP protocol is a stateless protocol. It means the client sends a request, the server responds the request, and the connection is closed at the response's end. But what can we do to keep the connection open? How can we receive messages from the server without sending another request to ask the server about new information? One good scenario of this problem is a chat app. When a user sends a message, all the users registered to the listener the message needs to be notified about the new message. How to do that using the HTTP protocol?

In the past, the common solution was implementing the long pooling approach in the server and keep the connection open. Later, the WebSocket came to solve this problem, but it is not an HTTP solution, and it can impact many points that are expecting an HTTP conversation. One example is, when you are using the reverse proxy or/and load balancer in front of your application, you'll need a special configuration to use WebSocket. Thinking in it the **Server-Sent Events** was created.

The SSE permit registering clients to listener events from the server. The SSE permit the client to open a connection and keep it opened to listen to all event or messages from the server. With this, using the example of the chat app, the SSE permits a client to register to receive messages from the server, without make a new request to the server. It is very good, but how can we use the SSE with Jakarta RESTful Web Service?

To show the example, I will create a Jakarta RESTful resource to the chat inside the **samplerestful** project — that exposes the resources. Just the Jakarta RESTful resource is sufficient to work with SSE and you need to code the client to connect to them using JavaScript, Java, or another language. As the JavaScript is not in the scope of this book, I'll connect to the chat resource using the Jakarta RESTful client inside the Thus, in our scenario, we'll send a message to **samplerestfulclient** that you are forwarding the message to **samplerestful** that will keep the connection with **samplerestfulclient** opened. *Figure 4.1* shows the scenario as follows:

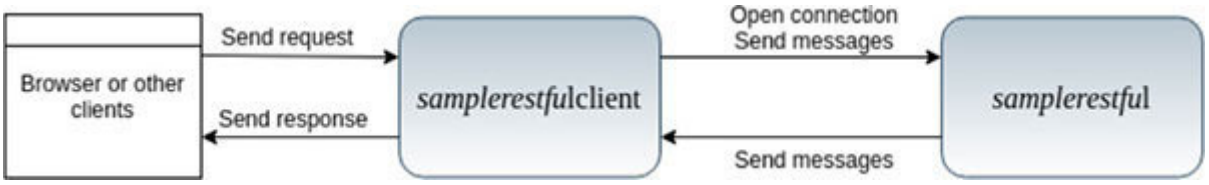

Figure 4.1: *Request Flow*

Note that this example is just academic, as we want to use Java and Jakarta EE in the entire example.

Thus, first, we'll create the **ChatResource** class inside the samplerestful project, shown as follows:

```
import

import jakarta.inject.Singleton;
```

```
import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.QueryParam;

import javax.ws.rs.Consumes;

import javax.ws.rs.FormParam;

import javax.ws.rs.POST;

import javax.ws.rs.PathParam;

import jakarta.ws.rs.core.Context;

import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

import jakarta.ws.rs.sse.Sse;


import jakarta.ws.rs.sse.SseBroadcaster;
```

```
import jakarta.ws.rs.sse.SseEventSink;


@Produces(MediaType.APPLICATION_JSON)

@Consumes(MediaType.APPLICATION_JSON)

@Singleton

{

@Context

private Sse sse;

private SseBroadcaster sseBroadcaster;

@PostConstruct

public void

        sseBroadcaster = sse.newBroadcaster();

    }

@GET
```

```
@Produces(MediaType.SERVER_SENT_EVENTS)

public void SseEventSink eventSink){

to

        sseBroadcaster.register(eventSink);

    }

@POST

@Consumes(MediaType.APPLICATION_FORM_URLENCODED)

public Response String message){


        sseBroadcaster.broadcast(sse.newEvent(message));

return Response.ok().build();

    }

}
```

The **ChatResource** is a singleton as it has the **SseBroadcaster** instance to send the message to all the clients registered to it.

You can have many **SseBradcaster** instances to send the messages to some kind of group.

The method **startConnection()** is used to register a new client, that is represented by the **SseEventSink** object. This method should be configured with as it is a call to register the client to the listener events, as follows:

@GET

public void SseEventSink eventSink){

to

}

The method **sendMessage()** adds a new message as an event to **SseBroadcater** object, that sends the event to all the registered clients, as follows:

@POST

public Response String message){

```
}
```

Now we will create the classes needed in **samplerestfulclient** to communicate with **samplerestful** via HTTP using the SSE.

First, we need to create the class which will be responsible to make the integration using the Jakarta RESTful client and the SSE feature to keep the connection open. The **ChatService** will be a CDI bean session scoped, which means the SSE connection will be kept open while the session exists. The **ChatService** class is shown as follows:

```
import jakarta.annotation.PostConstruct;

import jakarta.enterprise.context.SessionScoped;

import jakarta.ws.rs.client.Client;

import jakarta.ws.rs.client.ClientBuilder;

import jakarta.ws.rs.client.Entity;

import jakarta.ws.rs.client.WebTarget;

import jakarta.ws.rs.core.Form;
```

```
import jakarta.ws.rs.core.Response;

import jakarta.ws.rs.sse.SseEventSource;

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

@SessionScoped

{

String TARGET_SERVICE=

private SseEventSource sseEventSource;

private List messages;


@PostConstruct

        messages = new ArrayList<>();

    }
```

```
        Client client = ClientBuilder.newClient();

        WebTarget webTarget = client.target(TARGET_SERVICE);

        sseEventSource = SseEventSource.target(webTarget).build();

        sseEventSource.register(inboundSseEvent ->
messages.add(inboundSseEvent.readData()));

        sseEventSource.open();

    }

message){

        Form form = new Form();

message);

        Client client = ClientBuilder.newClient();

        Response response = client

                .target(TARGET_SERVICE).request()

                .post(Entity.form(form));
```

```
return response.getStatus() == Response.Status.OK.getStatusCode();

    }


        sseEventSource.close();

    }

public List

return messages;


    }

}
```

Note that the **ChatService** has the method called which is responsible for opening the SSE connection and keep it open, as follows:

```
        Client client = ClientBuilder.newClient();

        WebTarget webTarget = client.target(TARGET_SERVICE);
```

```
        sseEventSource = SseEventSource.target(webTarget).build();

-> messages.add(inboundSseEvent.readData()));

        sseEventSource.open();

    }
```

The **sendMessage()** method is a common **HTTP POST** request sending the message to be included as an event and is sent to all the registered clients. This method does not have anything special. Look at the following example:

```
message){

        Form form = new Form();

message);

        Client client = ClientBuilder.newClient();

        Response response = client

                .target(TARGET_SERVICE).request()

                .post(Entity.form(form));
```

```
return response.getStatus() == Response.Status.OK.getStatusCode();

    }
```

Now, to test, we will call the **samplerestfulclient** using the following **curl http://localhost:8080/samplerestfulclient-1.0-SNAPSHOT/resources/chat -v -c cookie.txt**

Note that we'll use the **-c cookie.txt** to keep the session. The following is the output:

```
> GET /samplerestfulclient-1.0-SNAPSHOT/resources/chat HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
* Added cookie
JSESSIONID="2ZYheMjry1nLtHchgy9mlnKy3MMszt1bQf6bX6rZ.localhost" for domain localhost, path /samplerestfulclient-1.0-SNAPSHOT, expire 0
< Set-Cookie:
JSESSIONID=2ZYheMjry1nLtHchgy9mlnKy3MMszt1bQf6bX6rZ.localhost; path=/samplerestfulclient-1.0-SNAPSHOT
< Content-Length: 0
```

Now you can add a new message. To do that, execute the following **curl** command:

curl -X POST http://localhost:8080/samplerestfulclient-1.0-SNAPSHOT/resources/chat -d "message=Test curl" -v -b cookie.txt

The following is the output:

```
> POST /samplerestfulclient-1.0-SNAPSHOT/resources/chat HTTP/1.1

> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 17
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 17 out of 17 bytes
< HTTP/1.1 200 OK
< Connection: keep-alive
* Added cookie JSESSIONID="o6sNXEBG8TnViyrqiihRO-1OJI2OFScqeeU3mwdU.localhost" for domain localhost, path /samplerestfulclient-1.0-SNAPSHOT, expire 0
< Set-Cookie: JSESSIONID=o6sNXEBG8TnViyrqiihRO-1OJI2OFScqeeU3mwdU.localhost; path=/samplerestfulclient-1.0-SNAPSHOT
< Content-Length: 0
```

Now we can list the messages by running the following **curl** command:

curl http://localhost:8080/samplerestfulclient-1.0-SNAPSHOT/resources/chat/messages -v -b cookie.txt

The following is the output:

> GET /samplerestfulclient-1.0-SNAPSHOT/resources/chat/messages HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Cookie: JSESSIONID=GkslHGv37YJZBfq0C04ABK16BLh0K3fsYDI6Cwiy.localhost
>

< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/json
< Content-Length: 32
*** Connection #0 to host localhost left intact**
["welcome to chat!","Test curl"]

As you can see, the messages are **to** and These messages are broadcasted to all the registered clients.

Conclusion

The Jakarta RESTful Web Service is a very important specification to the Jakarta EE ecosystem, mainly because the RESTful is used to integrate the microservice and to expose the APIs to the integrations. As we saw in this chapter, the Jakarta RESTful Web Service has many features to use the RESTful style working in an easy way. It provides a way to create a client and make an asynchronous communication between the services.

In the next chapter, we will show you what the Jakarta Enterprise Bean is and how we can use them to create good projects using good practices.

CHAPTER_5

Jakarta_Enterprise_Bean

Introduction

In this chapter, we will explain what Jakarta Enterprise Bean is, covering the principal concepts and practical. Besides, we'll explain how to implement a stateless bean, stateful bean, and singleton bean. Later, we will show you how the Jakarta Enterprise Bean manages the transaction, how we can schedule the tasks, and how we can use the **Message-Driven Beans** After studying this chapter, you will be able to use the Jakarta Enterprise Bean to solve the business problems and create business logic in your project.

Structure

In this chapter, we will discuss the following topics:

Understanding the Jakarta Enterprise Bean

Understanding the stateless bean, stateful bean, and singleton bean

Understanding how the Jakarta Enterprise Bean manages the transaction

Creating a schedule

Creating message-driven beans

Objectives

The main objectives of the chapter are to enable the readers to use the Jakarta Enterprise Bean in a project, create a stateless bean, create a stateful bean, create a singleton bean, and know when to use each of them. Furthermore, the readers will be enabled to manage the transactions with Jakarta Enterprise Bean, create schedules, and create the message-driven beans.

Understanding the Jakarta Enterprise Bean

As written in *Chapter 1, Introduction to Jakarta* Jakarta EE is multitier, and in general, it has three tiers – presentation tier, business tier, and integration tier. The business tier is the tier that has the business logic or business rule. The Jakarta Enterprise Bean, denoted in the past as EJB, is the Jakarta component provided to care about the business aspect and separate the business rule from the other tiers. The Jakarta Enterprise Beans provides many features to treat the problems about the business rule. As an example, we can quote the transaction. Generally, a business rule should run as a transaction implementing the all-or-nothing concept. Note that when I'm talking about the transaction, I'm not talking about just the database transactions, but the transactions that can include the other components, like messaging.

The Jakarta Enterprise Bean has a container that provides many kinds of abstractions, permitting us to delegate some questions that are not related to the business rules to the container. With this, we can care more about the business rules. Another example is when we need to schedule a task to execute. Without the Jakarta Enterprise Bean, we need to create some schedule using something like **cron** and make the logic to treat the concurrency using the file system. Using the Jakarta Enterprise Bean, we can create the schedule using just an annotation and we can create the class as a singleton bean. Thus, the Jakarta Enterprise Bean will manage the calls and the concurrency. As you can see, the

Jakarta Enterprise Bean has many interesting features that will be explained in more detail throughout this chapter.

Session beans

The session beans are the classes managed by the Jakarta Enterprise Bean container that encapsulates the business logic that can be invoked by a client (another Jakarta EE component or simple classes). The session bean can be of three types – stateless, stateful, and singleton.

The stateless session bean does not keep the conversation state. Thus, the developer should not keep the states in this kind of session bean, mainly because the instances of the stateless are pooled and it can cause some problems. We'll talk a little more about the stateless in the following section.

The stateful, different than stateless, is used when the developer wants to keep the conversation state between the client and the session bean. Note that the concept of the client here is a Jakarta EE component or simple classes that invoke the session bean. The conversation state will be kept while the client has the reference to the instance of the session bean. We'll talk a little more about the stateful in the following section.

The singleton is an interesting kind of session bean because it implements the singleton pattern, which means it will be only one instance in the entire application. Throughout this chapter, we'll see that the singleton is very similar to the stateless when we are talking about the functionality, but the difference is that the

singleton has only one instance of the session bean to the entire application.

Note that the session bean scope is managed by the Jakarta Enterprise Bean container and the developer does not keep it, but the developer should know it to use the session bean in a better way.

Now that we have overviewed the session bean, you can question about the message-driven beans. Why don't we quote the message-driven bean as a session bean? Is the message-driven bean a session bean? The answer is no, it is not a session bean. Different from the session bean, the message-driven beans are not called by a client. We'll see it in more detail throughout the chapter as well, but the main information here is, using the session bean, the client invokes it programmatically. Thus, the message-driven bean is called by the container, and it cannot be looked up using **Java Naming and Directory Interface** nor using the CDI.

Understanding the stateless bean, stateful bean, and singleton bean

In this section, we'll explain each session bean and how these work in more detail. Knowing how these session beans work is very important in creating a good project using the Jakarta Enterprise Bean. The first thing to know about the session beans is that these are managed by the Jakarta Enterprise Bean's container, which means, it should not be managed by another mechanism. It's the motive that you should not use the session bean as a class to be persisted or used as an entity. The session bean should be just a wrap of the business logic and be managed by the Jakarta Enterprise Bean's container.

After starting to explain each session bean, we'll create a project that will be used to create the examples described here. First, we'll create a new project called **samplejakartaenterprisebean** using the archetype described in *Chapter 1, Introduction to* Thus, to create a project you can use the Maven command as follows:

mvn archetype:generate -DarchetypeGroupId=net.rhuanrocha -DarchetypeArtifactId=jakartaee9-war-archetype -DarchetypeVersion=1.0.SNAPSHOT -DgroupId=net.rhuanrocha -DartifactId=samplejakartaenterprisebean

Note: You can choose another -DgroupId and/or -DartifactId.

Now, let's dive deep into each of these session beans in the following sections.

Stateless bean

As said earlier, the stateless beans are session beans created to process the task that does not keep the states in the class. You can create some properties in the session bean and keep the states, but the stateless does not guarantee that all the calls to the stateless bean will be processed in the same instance. With the stateless bean, each call can be processed by different instances, that are pooled to wait for the next call. Besides, it does not guarantee the same state to the same client, the state of the stateless can be shared by multiples clients. It is dangerous. So, never keep the states in a stateless bean.

When we create a stateless bean, the Jakarta Enterprise Bean creates the instances of the stateless on demand by the container and puts these to a pool of instances to be reused. These instances keep waiting for the call to process the tasks. After the call, the stateless comes back to the pool of instance, to be reused by another client. This makes sense, as the stateless are built to be used when the developer doesn't want to keep the states. A motive to keep the instances in a pool is the performance, as the instance will be reused, and the application does not need to initialize a new instance. Furthermore, it avoids an as you know the exact amount of instance will be created. Thus, instead of an overload generating an probably it will generate a performance issue, that tends to be less serious.

In general, the Jakarta EE platform provides a way to check the statistics of the pool of instances, and in case of a performance issue, the number of instances in the pool can be upgraded. In general, the number of instances is defined based on the CPU available, but it can be changed.

Creating a stateless bean is very easy, as you just need to annotate the class with the Look at the following example:

```
@Stateless


public String

return "Hello World";

    }

}
```

Here, the **HelloWorld** class can be looked up via JNDI or be injected using CDI, that is the modern approach. The Jakarta Enterprise Bean has great integration with CDI, and it can be injected, or can be a or the other CDI features.

Now, we'll create an example using the stateless, where the application will persist some products to the database. Note that it is a business logic, and this business logic doesn't need to keep the state in the class. To persist the data, we'll use the

Jakarta Persistence, but this chapter will not explain in detail the Jakarta Persistence, as we have a chapter focused on it.

Firstly, to work with Jakarta Persistence, we'll create a **persistence.xml** inside the The following is the **persistence.xml** used:

https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"

>

/>

As we are using the Wildfly in our examples, we'll use the **ExampleDS** data source that is the default data source of the Wildfly. It uses an in-memory database called The JNDI to this data source is

Now, we'll create an entity called **Product** to map the product to the database. The following is the entity:

```
import jakarta.persistence.Column;

import jakarfa.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Id;

import java.math.BigDecimal;

import java.util.Objects;

@Entity

{

@Id

@GeneratedValue

id;

@Column

private String name;
```

```
@Column(scale = 2)

private BigDecimal price;



getId() {

return id;

    }

public void id) {

= id;

    }

public String getName() {

return name;

    }

public void setName(String name) {

= name;
```

```
    }

public BigDecimal getPrice() {

return price;

    }

public void setPrice(BigDecimal price) {

= price;

    }

@Override

public boolean equals(Object o) {

if == o)

if (o == null || getClass() != o.getClass())

        Product product = (Product) o;

return Objects.equals(id, product.id);

    }
```

```
@Override

public int hashCode() {

return Objects.hash(id);

    }


}
```

Now we can create the stateless bean. The stateless bean will be called and as the name suggests, it is the class that wraps the business logic of the product domain. It follows the Business Object pattern. We have the **ProductBusiness** as follows:

```
import jakarta.ejb.Stateless;

import jakarta.persistence.EntityManager;

import jakarta.persistence.PersistenceContext;

import java.util.List;

@Stateless

{
```

```
@PersistenceContext

private EntityManager entityManager;

public List

return entityManager

p from Product



    }

product){

        entityManager.persist(product);

    }

public Optional id){

return

    }

}
```

As you can see, to turn a class into a Jakarta Enterprise Bean, you just need to annotate the class with It turns this class into a stateless bean, which means that it is managed by the Jakarta Enterprise Bean's container and gains the features provided by it. Using the CDI, it can now be injected to be used. You can use the JNDI to look up the stateless bean, but it is not so common today. It was common when it worked with the remote approach, but today it is an optional feature of the Jakarta Enterprise Bean, and the vendors don't need to implement it. It makes sense, as today we have many other ways to integrate an application that is more evolved. It is why we won't talk about the remote approach in this book.

The **ProductBusiness** is a stateless bean, and it has the logic to read and write the product to the database. As you already know, the Jakarta Enterprise Bean manages the transaction, and it means that the Jakarta Enterprise Bean is responsible to commit or rollback the translation according to the call's state. The rule to define when it should commit or rollback the transaction will be shown throughout this chapter, and we have a section focused on it. Note that the transaction here is not just with the database, but with any Jakarta resource that is able to participate in the transaction. One example is the Jakarta Messaging, which participates in the transaction as well and could be affected by the commit or rollback.

Now, to call the stateless bean, we will create a Jakarta RESTful resource that will be the stateless bean as the dependency injected by the CDI. Note that the Jakarta RESTful resource is

considered as the client from the perspective of the Jakarta Enterprise Bean, as explained earlier. The following is the code of the Jakarta RESTful resource:

```
import jakarta.inject.Inject;

import jakarta.ws.rs.GET;

import jakarta.ws.rs.Path;

import jakarta.ws.rs.Produces;

import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;


{

@Inject

private ProductBusiness productBusiness;

@GET

@Produces(MediaType.APPLICATION_JSON)
```

```
public Response findAll(){

return Response.ok(productBusiness.findAll()).build();

    }

@POST

@Consumes(MediaType.APPLICATION_JSON)

public Response save(Product product){

guarantee a new item will be created on the database

        productBusiness.save(product);

return Response

                .build();

    }

@GET


@Produces(MediaType.APPLICATION_JSON)
```

```
public Response id){

        Optional product = productBusiness.findById(id);


return Response

                        .status(Response.Status.NOT_FOUND)

                        .build();

        }

return

    }

}
```

As you can see, it has two methods, one to save a new product and another to list all the products. To test the endpoint to find all you can, use the following curl's command:

curl http://localhost:8080/samplejakartaenterprisebean-1.0-SNAPSHOT/resources/products -v

The following is the output:

```
> GET /samplejakartaenterprisebean-1.0-
SNAPSHOT/resources/products HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/json
< Content-Length: 2
* Connection #0 to host localhost left intact
[]
```

To test the endpoint to save, you can use the following curl's command:

```
curl -X POST http://localhost:8080/samplejakartaenterprisebean-1.0-
SNAPSHOT/resources/products -H "Content-Type: application/json"
--data '{"name":"notebook","price":1000}' -v
```

The following is the output:

```
> POST /samplejakartaenterprisebean-1.0-
SNAPSHOT/resources/products HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
```

> Content-Type: application/json
> Content-Length: 32
>
*** upload completely sent off: 32 out of 32 bytes**
< HTTP/1.1 201 Created
< Connection: keep-alive
< Location: http://localhost:8080/samplejakartaenterprisebean-1.0-SNAPSHOT/resources/products/1
< Content-Length: 0

As you can see, the product was saved, and you did not care about commit and rollback or the lifecycle of the stateless instance. After executing, the instance will come back to the pool to be reused by another request. Note that we didn't keep the states in the stateless bean.

Stateful

In the real world, many a times, the application should keep the states cross-requesting to make its tasks. An example is the shopping cart that should be kept. You can keep it in the frontend or in the backend. In case of the backend, the stateful bean is a good option to do it.

With the stateful bean, the instance has a relationship of one-to-one with the client. It means that the instance will not be reused by another client and the state will be kept while the client keeps the instance live. Thus, the stateful bean does not work with the pool of instance, but after the client drops the instance, this instance is killed/removed. Thus, to show you an example of how to use the stateful bean, we'll improve the **samplejakartaenterprisebean** and add the shopping cart feature to it. Inside the shopping cart, we will have the products already represented in our applications as The shopping cart will have the option to add, remove, clean, and finalize them. In the finalize stage, a registry of purchase will be saved in the database. Thus, to improve our test to attend this scenario, first we'll create an entity called which will map the purchase to the table in the database, as shown in the following code snippet:

```
import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;
```

```
import  jakarta.persistence.Id;

import  jakarta.persistence.ManyToMany;

import  java.util.List;

import  java.util.Objects;

@Entity

{



@Id

@GeneratedValue

id;

@ManyToMany

public  List  products;

getId()  {

return  id;
```

```
    }

public void id) {

= id;

    }

public List getProducts() {

return products;

    }

public void setProducts(List products) {

= products;

    }

@Override

public boolean equals(Object o) {

if == o)

if (o == null || getClass() != o.getClass())
```

```
        Purchase purchase = (Purchase) o;

return Objects.equals(id, purchase.id);

    }

@Override

public int hashCode() {

return Objects.hash(id);

    }

}
```

Now, we will create a stateless bean that will be responsible for the logic about the purchase, as shown in the following code snippet:

```
import jakarta.ejb.Stateless;

import jakarta.persistence.EntityManager;

import jakarta.persistence.PersistenceContext;
```

```
import java.io.Serializable;

import java.util.List;

import java.util.Optional;

@Stateless

{

@PersistenceContext

private EntityManager entityManager;

public List

return entityManager

p from Purchase p join fetch


    }

purchase){

        entityManager.persist(purchase);

    }
```

```
public Optional id){

return

    }

}
```

Note that this stateless implements the **Serializable** interface. The motive is that this class will be injected by a **SessionScoped** CDI bean, and it is a requirement to a Note that we need to update the **ProductBusiness** to implement the **Serializable** as well, as follows:

```
@Stateless

{

        ....

}
```

Now, we'll create the stateful bean called **ShoppingCartBusiness** which is responsible for wrapping the shopping cart's logic. Pay attention in this class, as it is the aim of this topic. The remaining classes are just to turn the test possible. The following is the code of the stateful bean:

```
import jakarta.annotation.PostConstruct;

import jakarta.ejb.Stateful;

import jakarta.inject.Inject;

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

@Stateful

{

@Inject

private PurchaseBusiness purchaseBusiness;

private List products;

@PostConstruct


= new ArrayList<>();
```

```
    }

product){




    }

product){

    }

    }

public List

    }

public Purchase

        Purchase purchase = new Purchase();
```

```
            purchase.setProducts(products);

            purchaseBusiness.save(purchase);

return  purchase;

      }

}
```

Note that this stateful bean has methods to add products, remove, and finalize the cart. When the cart is finalized, a new purchase with the products is created in the database. Note that this class expects to keep the state cross request. As we want the shopping cart to keep the states in the session scope, we will inject the stateful bean in a session scoped CDI bean. Remember that the session bean (from the Jakarta Enterprise Bean) does keep the states while the instance is kept referenced. Thus, if we inject it inside a session scoped CDI bean, the reference will be kept while the session is valid. So, now we will create the Jakarta RESTful resource called **PurchaseEndpoint** and annotate it with Look at the following example:

```
import  jakarta.enterprise.context.SessionScoped;

import  jakarta.inject.Inject;

import  jakarta.ws.rs.*;
```

```
import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

import java.io.Serializable;

import java.net.URI;

import java.util.Optional;


@SessionScoped

{

@Inject

private PurchaseBusiness purchaseBusiness;

@Inject

private ShoppingCartBusiness shoppingCartBusiness;

@Inject

private ProductBusiness productBusiness;
```

```
@GET

@Produces(MediaType.APPLICATION_JSON)

public Response findAll(){

return Response

                    .ok(purchaseBusiness.findAll())

                    .build();

    }

@PUT


@Consumes(MediaType.APPLICATION_FORM_URLENCODED)


public Response

        Optional product = productBusiness.findById(idProduct);


return Response
```

```
                    .status(Response.Status.BAD_REQUEST)

                    .build();

            }


return  Response.ok().build();

      }

@POST

public  Response  finalizePurchase(){


return  Response

                    .status(Response.Status.BAD_REQUEST)

                    .build();

            }

            Purchase  purchase  =  shoppingCartBusiness.finalizeCart();
```

```
return

    }

@GET


@Produces(MediaType.APPLICATION_JSON)

public Response id){


        Optional purchase = purchaseBusiness.findById(id);


return Response

                    .status(Response.Status.NOT_FOUND)

                    .build();

        }

return Response


                .build();
```

```
    }

}
```

As you can see, we have the **addProduct** method to add a product to the shopping cart and the **finalizePurchase** method to finalize the purchase. To test this example, you can apply the following curl's commands.

Note: The REST resource is supposed to be stateless, but just as an example, we created a resource with SessionScoped and kept the states inside the REST resource, but it should be avoided as it goes against the **REST concept.**

To add a product to the shopping cart, use the following command:

curl -X PUT http://localhost:8080/samplejakartaenterprisebean-1.0-SNAPSHOT/resources/purchases/shoppingcart -H "Content-Type: application/x-www-form-urlencoded" --data "idProduct=1" -v -c cookie.txt

The following is the output:

> PUT /samplejakartaenterprisebean-1.0-SNAPSHOT/resources/purchases/shoppingcart HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080
> Accept: */*
> Content-Type: application/x-www-form-urlencoded
> Content-Length: 11
>
*** upload completely sent off: 11 out of 11 bytes**
< HTTP/1.1 200 OK
< Connection: keep-alive
*** Added cookie JSESSIONID="AZKdD4O7yQfBnFfxym_Rak9CywlgoSYtEoMdv7U1.localhost" for domain localhost, path /samplejakartaenterprisebean-1.0-SNAPSHOT, expire 0**
< Set-Cookie: JSESSIONID=AZKdD4O7yQfBnFfxym_Rak9CywlgoSYtEoMdv7U1.localhost; path=/samplejakartaenterprisebean-1.0-SNAPSHOT
< Content-Length: 0

Note that you should use **-c** to keep the cookies and send the **idProduct** parameter, which represents the id of the product.

Now, you can use the following command to finalize the **-X POST http://localhost:8080/samplejakartaenterprisebean-1.0-SNAPSHOT/resources/purchases -v -b cookie.txt**

The following is the output:

> POST /samplejakartaenterprisebean-1.0-SNAPSHOT/resources/purchases HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*

> Cookie: JSESSIONID=AZKdD4O7yQfBnFfxym_Rak9CywlgoSYtEoMdv7U1.localhost
>
< HTTP/1.1 201 Created
< Connection: keep-alive
< Location: http://localhost:8080/samplejakartaenterprisebean-1.0-SNAPSHOT/resources/purchases/2
< Content-Length: 0

Now, you can list the purchase to check if it was persisted. To do that, you can use the following curl's command:

curl http://localhost:8080/samplejakartaenterprisebean-1.0-SNAPSHOT/resources/purchases -v -b cookie.txt

The following is the output:

> GET /samplejakartaenterprisebean-1.0-SNAPSHOT/resources/purchases HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Cookie: JSESSIONID=AZKdD4O7yQfBnFfxym_Rak9CywlgoSYtEoMdv7U1.localhost
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/json

< Content-Length: 66
< Date: Wed, 14 Oct 2020 22:40:41 GMT
<
*** Connection #0 to host localhost left intact**

[{"id":2,"products":[{"id":1,"name":"notebook","price":1000.00}]}]

As you can see, the shopping cart kept the product cross request and created a new purchase in the database when we finalized the purchase.

The stateful bean is a useful session bean, but it is recommended to be used just when you need the session bean to keep the state cross-calling, as the stateless bean tends to use fewer computer resources.

Singleton

The singleton pattern defines that just one instance will be created for the entire application. This pattern is implemented by the Jakarta Enterprise Bean, which guarantees just one instance by the application. The singleton bean works similar to the stateless bean regarding the reuse of the instances, but the difference is that the stateless bean has many instances and the singleton has just one instance. Thus, we can say that the singleton bean implements the singleton pattern, and the stateless bean implements the multiton pattern.

As the singleton has just one instance, it is perfect to keep the states that should be shared in the entire application. As an example, we can think about the scenario where we want to count the number of accesses to the application. Thus, for each access, we'll increment one to the count variable (count++). This is perfect for the singleton bean, as we need to guarantee that the same instance will be called to count all the accesses. Note that we need to care about the concurrent access to increment the variable. The singleton has an easy mechanism to control the concurrent access and manage the locks of the singleton instance.

To test the singleton bean feature, we'll add the feature of the access count to our application. Thus, each access will be incremented, and we will be able to consult the access count via an HTTP request. Thus, to implement this feature, first, we'll

create the **AccessSingleton** to our application. Look at the following example:

```
@Singleton



@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)

{

private Integer count;

@PostConstruct

public void


    }

@Lock(LockType.WRITE)

public void increment(){


    }

@Lock(LockType.READ)
```

```
public Integer getCount(){

    }

}
```

Thus, to create the singleton, we need to just annotate the class with You can note that we used the **@ConcurrencyManagement** to tell that the concurrency control will be managed by the container. It's the default value, but we typed it to show you how to define it. The other option is the concurrency managed by the bean, which means that the developer will care about the concurrency. The increment method was annotated with which means that the instance will be locked when the increment method is called. The **getCount** method is annotated with which means that it will not lock the instance. Thus, the **@Lock(LockType.WRITE)** locks the instance and the **@Lock(LockType.READ)** does not lock.

Now, we'll create the endpoint called **AccessConutEndpoint** to consult the access count. This endpoint will increment the count as well. To count each access, the singleton should be used in each endpoint. An easy way to put it in all the endpoints is to create a CDI interceptor, but as we want to just show the feature, we'll not do it, and the increment method will be called just by The following is the **AccessCountEndpoint** code:

```
public class AccessCountEndpoint {

@Inject

    private AccessSingleton accessSingleton;

@GET

    public Response count(){




    }

}
```

To test it, you can use the following curl's command:

curl http://localhost:8080/samplejakartaenterprisebean-1.0-SNAPSHOT/resources/access -v

The output is as follows:

> GET /samplejakartaenterprisebean-1.0-SNAPSHOT/resources/access HTTP/1.1

> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 1
< Date: Mon, 19 Oct 2020 17:00:20 GMT
<
*** Connection #0 to host localhost left intact**
1

The singleton bean is started at the first access, but you can tell the application to start it in the startup time. It is a good feature when you have a singleton bean that is expensive to be created, and you want to save time in the runtime. To start the singleton bean at the startup time, you should annotate the singleton bean with Look at the following example:

```
@Singleton


@Startup

public class AccessSingleton {
```

....

}

Thus, the singleton will be created and will be available to be injected.

Understanding how the Jakarta Enterprise Bean manages the transaction

In the real world, many applications should execute the processes as a transaction, which means that it should execute the processes as a unique and isolated unit. In case of any error, all the steps done should be undone. The relational database provides a way to execute the write operations inside a transaction, and the database manages the consistency and isolation. The relational databases work with the ACID transactions, which means Atomicity, Consistency, Isolation, and Durability. The ACID concept is described as follows:

It is the everything or nothing concept. Either all the steps of the process inside the transaction are executed or nothing is executed.

It guarantees that the data are consistent according to the data roles. In case of any inconsistency detected, the entire transaction is undone (or rolled back).

A running transaction should not be interfered by another transaction. In case of multiple transactions running concurrently, it should be serialized, and one transaction should not interfere with another one.

The data already validated should be persisted and could not be lost in case of fails. Once the data is persisted, it should be kept

even in case of error.

The Jakarta EE extends this concept to the other Jakarta resources such as Jakarta messaging, permitting another Jakarta resource to participate in a transaction in an easier way. It is provided by the Jakarta transaction specification, and the Jakarta Enterprise Bean has a way to abstract the transaction management from the developer. It means that the developer does not need to manage the transaction by yourself but can delegate it to the Jakarta container.

The Jakarta Enterprise Bean has two ways to manage the transactions, that is the **Container-Managed Transaction** or **Bean-Managed Transaction** In this chapter, we'll focus on the CMT as it is more common to be used, but we'll show you how to use the BMT as well.

As the name suggests, in the CMT, the transactions are managed by the Jakarta container and the developer does not need to apply commit and rollback, nor start the transaction. In the BMT, the transactions are managed by the developer and the developer should start the transaction and apply the commit or roll back. Note that to us, the transaction management means to define the scope of the transaction and decide if the changes should be persisted or not.

Container-managed transaction

The **container-managed transaction** is the default management transaction of the Jakarta Enterprise Bean. It means whether or not the developer defines this configuration to the session bean, it will be configured as CMT. With the CMT, the transaction is managed by the container that defines the transaction's scope according to the configuration defined to the session bean. All the configurations are defined with the annotation, that makes it very easy. As we said, the default transaction management is the CMT, but if you want to configure it by yourself, you can use the annotation **@TransactionManagement** in the session bean. Look at the following example:

```
@Stateless

TransactionManagementType.CONTAINER)

public class PurchaseBusiness implements Serializable {

        ...

//attributes and methods

}
```

Thus, the session bean was configured with CMT. Now we should configure how the container will define the scope of the transaction. It is done defining the transaction attribute to the class or method annotating them. If you define it in both the class and the method, the method overrides the configuration of the class. The transaction attribute can be of six types – and The default value is which means that if it is not defined, the transaction attribute will be the Each of these transaction attributes are explained as The method invocation should happen inside a transaction context, which means that a transaction should already be started. In case the invocation is not inside a transaction context, an exception will be launched.

The method should not be invoked inside a transaction context, which means a transaction should exist when the method is invoked. In case of the invocation being inside the transaction context, an exception will be launched.

When the method is invoked inside a transaction context, the transaction is suspended, and the method is executed outside a transaction context. At the end of the method execution, the transaction is resumed.

When the method is invoked inside a transaction context, it is executed inside the transaction context without a problem. In case of the invocation outside a transaction, a new transaction will be started.

When the method is invoked inside a transaction context, the current transaction is suspended, and a new transaction is started.

At the end of the method, the transaction is committed/rolled back, and the transaction suspended is resumed. In case of the invocation outside a transaction context, a new transaction is started normally.

When the method is invoked inside a transaction context, the method is executed inside the activated transaction context. In case of the invocation outside the transaction context, the method is executed outside the transaction without a problem.

Thus, you can configure any of the transaction attributes described in the preceding section to the session bean. To do this, you just need to configure the **@TransactionAttribute** annotation according to the transaction attribute you want. The following is an example which shows configuring the transaction attribute to

@Stateless

TransactionManagementType.CONTAINER)

public class PurchaseBusiness implements Serializable {

...

purchase){

```
    }

    ...

}
```

As you can note, the method save was configured to You can configure it to the class as well. Remember the configuration to the method has priority over the class.

Commit and rollback

The transaction is rolled back or committed when the method that started the transaction ends the processing. Thus, when the method ends, the Jakarta Enterprise Bean context should define if the transaction should be committed or rolled back. To define it, the Jakarta Enterprise Bean checks if the executing has generated some exception, and if generated, what is the type of exception.

In Java, we have two types of exceptions, i.e., checked exceptions and unchecked exceptions. In practice, the unchecked exception extends **RuntimeException** and the checked exception does not extend it. When the system generates an unchecked exception, the transaction is rolled back at the end of the method's execution. Otherwise, it commits the transaction. The other scenario of the rollback is in the case of some issue to commit, such as a timeout or some problem or unavailability of the database or resource. Thus, all **RuntimeException** generates a rollback. Note that the rollback is applied when the unchecked exception is generated and not treated. If the exception is treated in a then the transaction should be committed. We can change this behavior and annotate the exception class with the If you annotate a checked exception with this annotation and configure the rollback attribute to true, the exception will generate a rollback as well.

Creating a schedule

Many a times, the application needs to execute some task periodically (per hour, day, month, or year) and this task is not started by the user, but by the application. With this, we need a way to schedule a task to be executed according to a time, but executing the tasks have many questions that the developer should think about. For example – What is done in case of failure? How to treat concurrency?

In general, the tasks scheduled are long-running tasks and it is possible that the other task can be started before the previous task finishes. It must be thought out.

The Jakarta Enterprise Bean has a feature called **Timer** that permits us to schedule the tasks to be executed and the execution is managed by the Jakarta container. Furthermore, it can be executed as a singleton bean, taking the advantage of the lock of singleton. With this, the Jakarta container will manage the execution and the concurrency as well. The Jakarta Enterprise Bean provides a way to create a schedule using the annotation approach or programmatically. To show this feature, we will create a schedule that generates a simple log in every 10 minutes. The following is the class called **HelloSchedule** that has the method **logHello** and is scheduled to execute in every 10 minutes:

```
import jakarta.ejb.Schedule;
```

```
import jakarta.ejb.Singleton;

import java.util.logging.Logger;

@Singleton

{

private Logger logger =



= =



    }

}
```

Firstly, to schedule a task, you configure the class as a session bean, that can be a singleton, stateless, or stateful. In general, we use the singleton as it just needs one instance of the object, and it permits you to configure the lock mode. After that, you should annotate the method that will be scheduled. Note that we configure the hour to "*", which means that it will be executed for all hours, and we configured the minute "0,10,20,30,40,50", which means that it will execute in every 10 minutes. If we

configure the minute to “*”, it will be executed every hour and every minute. You can configure the second, minute, hour, dayOfMonth, month, dayOfWeek, year, timezone, info, and persistent parameters to the schedule.

Persistent schedule

The Jakarta Enterprise Bean timer can be of two types, persistent and non-persistent. In the persistent timer, that is the default, the timer is persisted in the disk or database and the timer survives to restart. Thus, in a persistent timer, if the task is scheduled and the server is restarted, the task will not be lost and will be executed. In the non-persistent timer, the time is kept in the memory, and in case of restart, the task is lost.

Scheduling programmatically

Another way to schedule the tasks is programmatically instead of using annotation. It is good when we want to schedule it according to some logic. To schedule a task programmatically, we'll create a singleton bean called The following is the class example:

```
import jakarta.annotation.PostConstruct;

import jakarta.annotation.Resource;

import javax.ejb.ScheduleExpression;

import javax.ejb.Singleton;

import javax.ejb.Startup;

import javax.ejb.TimerService;

import javax.ejb.Timeout;

import javax.ejb.Timer;

import java.util.logging.Logger;
```

```
@Singleton

@Startup

{

private Logger logger =

@Resource

private TimerService timerService;

@PostConstruct

        ScheduleExpression expression = new ScheduleExpression();




        timerService.createCalendarTimer(expression);

    }

@Timeout
```

```
timer) {

programmatically

    }

}
```

Thus, as you can see in the preceding example, we created a singleton bean configured to be started at the startup time, and in this, the method **init** configured as **PostContruct** is executed, creating the expression and schedule calling the **timerService.createCalendarTimer(expression)** as follows:

```
@PostConstruct

        ScheduleExpression expression = new ScheduleExpression();

        timerService.createCalendarTimer(expression);

    }
```

When it reaches the time configured, the method annotated with **@Timeout** is executed. Thus, the method executed in a schedule

in the programmatic way will be the method annotated with As you can see in the following example, it generates a log as well:

@Timeout

timer) {

programmatically

}

As you can see, you can define the expression used according to some logic in a dynamic way.

Understanding how we can create a message-driven beans

As you will see, the Jakarta Enterprise Bean has a good integration to Jakarta messaging that permits us to consume a queue or topic in an easier way. However, this chapter will not explain the Jakarta messaging as we have a chapter that is focused on it. Thus, this section will show you just how to use the **Message-Driven Beans**

Sometimes, we need to consume some queue or topic of the Jakarta messaging but continue listening to a queue or topic can be a non-trivial task for the developer. To make it easier to continue listening to a queue or topic, the MDB was created. With MDB, you can configure a class to react to a new message in a queue or topic. With this, for each new message, this class will be executed to process the message. To show you how to create an MDB, we will create an MDB called **HelloMDB** that will get a message from the queue and will log it. Look at the following example:

```
import jakarta.annotation.Resource;

import jakarta.ejb.ActivationConfigProperty;

import jakarta.ejb.MessageDriven;

import jakarta.ejb.MessageDrivenContext;
```

```
import jakarta.jms.MessageListener;

import java.util.logging.Logger;

= {

=

                    propertyValue =


=

                    propertyValue =

})

{

private Logger logger =

@Resource

private MessageDrivenContext mdc;

@Override
```

```
message) {

        logger.info(message.toString());

//business logic

    }

}
```

Note that to create an MDB, the class should extend **MessageListener** and should be annotated with **@MessageDriven** that will have the configurations needed for the MDB to know the queue or topic to connect. The **@MessageDriven** is configured with two properties that is the **destinationLookup** with the JNDI to the queue and the **destinationType** with the type of the destination (queue or topic). When the **java:/jms/queue/MyQueue** queue has a message, it is delivered to the **onMessage** method that executes the logic. The Wildfly is not in the scope of this book and all the techniques taught here will work in any implantation compliance to Jakarta EE. But as showed these examples using the Wildfly, I will share the command to be used to add the queue in the Wildfly. The following is the command:

jms-queue add --queue-address=MyQueue --entries=java:/jms/queue/MyQueue

Note that the entry informed is the same JNDI used by the MDB. The Jakarta EE uses the JNDI to lookup the queues and topics. Thus, you should configure the correct JNDI.

Conclusion

In this chapter, we showed the main concepts around the Jakarta Enterprise Bean and how to use it to provide good solutions to the business tier. The Jakarta Enterprise Bean is a very important API specification of the Jakarta EE. In this chapter, we talked about all the session beans, the transaction management, the timer, and MDB. In general, a Jakarta application uses the Jakarta Enterprise Bean to do something.

In the next chapter, we will talk about the Jakarta Persistence and how to use it. In this chapter, we mentioned the Jakarta Persistence, but not deeply. In the next chapter, we will dive deep into it.

CHAPTER 6

Jakarta Persistence

Introduction

In this chapter, we will explain what Jakarta Persistence is, covering the principal concepts and practical. Besides, we'll explain how we can make the operations in a relational database, how we can map the table to an entity, how we can use the JPQL to make queries, how can we use the EntityManager, how we can control the concurrent access to the entity, and how we can make the fetch plans using the Entity Graphs. After studying this chapter, you will be able to use the Jakarta Persistence to integrate your application to some relational database and learn how to make operations using it.

Structure

In this chapter, we will discuss the following topics:

Understanding Jakarta Persistence

Creating an entity

Creating an entity relationship

Using JPQL

Controlling concurrent access to entity data

Creating fetch plans and entity graphs

Objectives

The main objectives of the chapter are to enable the readers to use the Jakarta Persistence in a better way to map the tables to the entities, create an entity relationship, and use the JPQL to query the data. Besides that, the reader will be able to control the concurrent access to the entity data and create fetch plans and entity graphs.

Understanding the Jakarta Persistence

In the real world, almost all applications integrate into some database. In the past, these databases were relational databases, and although this has decreased because NoSQL has increased in the market, the relational database is still widely used. We know that the relational database works with the tables and relations between these tables. However, the Java applications work in the **Object-Oriented Programming** paradigm, which is different from the relational paradigm. It generates the object-relational impedance mismatch.

Thus, it is very common to see a class of frameworks called **Object-Relational Mapping** that works as an adapter between those paradigms, mapping the tables and their relationships to the objects called Many languages have the ORM frameworks to solve it, and in the Jakarta umbrella, it has Jakarta Persistence.

The Jakarta Persistence is a specification to the ORM framework that provides a set of features to mapping, reading, and writing the data from the relational databases abstracting many difficulties of these operations. Because of this, the developer does not need to care about implementing these things and can therefore, care more about the business rule. Furthermore, in some cases, the use of Jakarta Persistence can improve the performance of the access to data, as it works with the first-level cache and second-level cache. It has done good work regarding the prepared

statement as well, caching the prepared statement to reuse. Although the Jakarta Persistence is a good solution and improves the performance in some cases, the developer should pay attention when they are using it and creating the mapping, because an issue in the mapping can decrease the performance a lot. Thus, I suggest you study this chapter very carefully.

The Jakarta Persistence is a tier over the JDBC, thus, in the background, the Jakarta Persistence executes the JDBC, but it is abstracted to the developer. Thus, we can split the Jakarta Persistence into three parts, which are as follows:

Entity

EntityManager

JDBC Driver

Although the JDBC is not provided and maintained by Jakarta Persistence, it is listed here because Jakarta Persistence uses it and you should select the JDBC driver according to the database. It is the JDBC driver that makes the communication with the database.

Entity

As discussed earlier, the tables should be mapped to the classes and the relationships should be expressed. To do that, the Jakarta Persistence has the entity, which is a representation of a table and its relationships. *Figure 6.1* is a graphical representation of this:

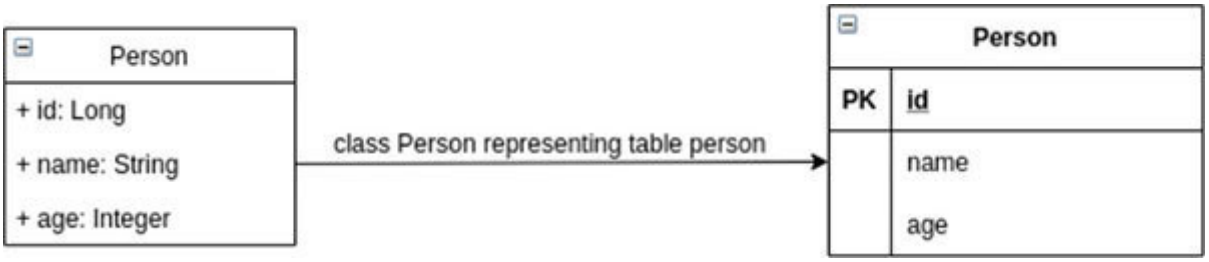

Figure 6.1: *Class mapping table*

The following is an example of the **Person** entity:

@Entity

@Id

id;

@Column

private String name;

@Column

private Integer age;

//getter and setters

}

As you can see, the class should be annotated with the the id should be annotated with and the columns, which are not relationships, should be annotated with *Table 6.1* is the list of the principal annotations to an entity, as follows:

follows:

follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows:
follows: follows:
follows: follows:

follows: follows:
follows: follows:

Table 6.1: *Relationship*

As you can see, the Jakarta Persistence represents the relationship with the annotations and The following is another example of an entity, but now it is representing many-to-many relationships:

@Entity

@Id

id;

@Column

private String name;

@ManyToMany

private List books;

//getter and setters

}

The book entity is as follows:

@Entity

@Id

id;

@Column

private String title;

...

//getter and setters

}

As you can see, it is easier to map the table to the class. If you try to do it yourself, you will see that it is very hard in some cases. It is just an introduction, and we'll see more of the entity throughout this chapter.

EntityManager

The **EntityManager** is the main element to apply the operations of reading and writing to the database and manage the entities. The **EntityManager** has a persistence context that a set of entities can participate in. The persistence context can be configured by the persistence-unit, which we'll see soon in this chapter. Thus, each **EntityManager** is associated with a persistence-unit that has a set of configurations to the persistence context. The set of configuration is defined by the **persistence.xml** inside the **META-INF** folder. It is very important to note that the Jakarta Persistence specification does not provide a way to externalize this configuration. It should change in the future as the externalization of configuration is important to the Cloud-Native Application. For now, the persistence.xml should be in META-INF. Thus, it should be placed as **src/main/resources/META-INF** or

To build an instance of the Jakarta Persistence has the **EntityManagerFactory** to read the persistence-unit and create an instance of The following is an example of building an EntityManager's instance using the

EntityManagerFactory factory = Persistence

EntityManager entityManager =factory.createEntityManager();

The following is an example of the

version="1.0" encoding="UTF-8" ?>

https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"

>

/>

/>

/>

/>

As you can see, the **persistence.xml** has the persistence-unit named jakartaee-unit that has the database configurations. You can use this to configure some datasource instead of configuring the database inside the In general, the data source is used as it creates a pool connection. Note that I'm using the **transaction-type="RESOURCE_LOCAL"** because I'm using the **EntityManagerFactory** to get an The default is JTA, but you cannot use JTA and the **EntityManagerFactory** together. We'll see this in the following sections in our scenario of the test.

Note that the JDBC driver is configured at the If you use some datasource, it will be configured in the datasource instead of The following is the property that configures the JDBC driver:

name="jakarta.persistence.jdbc.driver"
value="org.postgresql.Driver" />

To show you a clear example of the Jakarta Persistence uses, we will create a scenario of books and authors. Thus, in this scenario, we will have a many-to-many relationships between the authors and the books, and we'll have the operations to write and read the data to these tables/entities. With this, we'll create a new project called **samplejakartapersistence** using the archetype described in *Chapter 1, Introduction to* to this scenario. The following maven command could be used to it:

mvn archetype:generate -DarchetypeGroupId=net.rhuanrocha -DarchetypeArtifactId=jakartaee9-war-archetype -

DarchetypeVersion=1.0.SNAPSHOT -DgroupId=net.rhuanrocha -DartifactId=samplejakartapersistence

Note that you can choose another **-DgroupId** and/or

As we are using the Wildfly, we will use the datasource provided by the Wildfly that uses the H2 database, that is an in-memory database. But you can create another database to connect to another kind of database. As the Wildfly is not in the scope of this book, we will not show you how to create a new datasource, and we'll use the one that is already provided. The JNDI of the datasource provided is Thus, we should configure it into persistence.xml. With this, we'll create the **persistence.xml** inside the **WEB-INF/classes/META-INF** with the following content:

https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"

>

/>

Note that we created a persistence-unit named and configured the datasource and a property called **jakarta.persistence.schema-generation.database.action=drop-and-create** which tells the Jakarta Persistence to drop and create the database and the table when the application starts.

Now we are able to create our scenario. Thus, we'll start by creating the entities.

Creating an entity

As discussed earlier, we should create a class to map a table and this class is called Thus, inside the scenario of the author and the book, we will create the Author entity as the first entity, and then we'll create the **Book** entity. The following is an example of the **Author** entity:

```
import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Id;

@Entity

{

@Id

@GeneratedValue

id;
```

@Column

private String name;

//getters and setters

}

As we can see, to create an entity, we should annotate the class with Thus, the table is mapped according to the class name, that is, the class **Author** is mapped to a table In case we need to map another table to the class author, we can use the **@Table** annotation. The following is an example:

@Entity

=

public class Author {

...

}

As you can see, the **@Table** maps the **Author** class to the **MyAuthor** table.

All the entities should have an attribute to be the ID, that is a representation of the primary key of the table. This attribute should be annotated with The **@GeneratedValue** tells Jakarta Persistence to auto generate the ID, that in general the value is incremented.

To map the column, we should annotate the attribute with Thus, the column is mapped according to the attribute name. To map the table's column without using the default value, you can configure the column's name with **@Column(name =**

Now, we should declare the entity in the persistence.xml. In case we are using Jakarta Persistence in a Jakarta EE application instead of in a plain vanilla Java SE application, as shown in *Chapter 5, Jakarta Enterprise* we don't need to declare the entities in the The following is an example:

https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"

>

/>

Creating an entity relationship

The Jakarta Persistence has four annotations to work with relationships, that is the and The following is a description of these annotations:

Used to declare the many-to-one relationship. The type of attribute annotated with this annotation should be an entity.

Used to declare the many-to-many relationships. The type of attribute annotated with this annotation should be a collection of an entity.

Used to declare the one-to-one relationship. The type of attribute annotated with this annotation should be an entity.

Used to declare the one-to-many relationships. The type of attribute annotated with this annotation should be a collection of an entity.

Now, we'll create the **Book** entity that has a relationship to the The following is the **Book** entity:

```
import jakarta.persistence.Entity;

import jakarta.persistence.Id;
```

```
import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Column;

import jakarta.persistence.ManyToMany;

import jakarta.persistence.FetchType;


@Entity

{

@Id


@GeneratedValue

    private Long id;

@Column

    private String title;

@Column
```

private String description;

@ManyToMany

private Set authors;

//getters and setters

}

As you can see, we used the same annotations as the but in the we have used the **@ManyToMany** annotation to map the many-to-many relationships of the **Book** and the Now, we should declare the **Book** entity in the The following is an example:

https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"

>

/>

Now, to test, we will create two CDI beans that will be used as a class responsible to treat the business rules. With this inside these CDI beans, we'll use the **EntityManager** to read and write the data to the database. Besides that, we'll create two Jakarta RESTful resources to test the Jakarta Persistence. Note that this chapter is not about Jakarta RESTful, and we'll not talk much about it here. Thus, the following are the descriptions of the classes used by us in this example:

CDI bean with the business logic about the author.

CDI bean with the business logic about the book.

Jakarta RESTful resource to receive the HTTP request about the author.

Jakarta RESTful resource to receive the HTTP request about the book.

Thus, the first class is The following is this class:

import net.rhuanrocha.entities.Author;

import jakarta.annotation.PostConstruct;

import jakarta.enterprise.context.ApplicationScoped;

import jakarta.persistence.EntityManager;

```
import jakarta.persistence.EntityManagerFactory;

import jakarta.persistence.Persistence;

import java.util.List;

@ApplicationScoped

{

entityManagerFactory;



public void

        entityManagerFactory = Persistence




    }

findAll(){

EntityManager entityManager =
entityManagerFactory.createEntityManager();
```

```
authors = entityManager

a from Author

                    .getResultList();

        entityManager.close();

return authors;

    }

save author){

EntityManager entityManager =
entityManagerFactory.createEntityManager();

        entityManager.getTransaction().begin();

        entityManager.persist(author);

        entityManager.getTransaction().commit();

        entityManager.close();

return author;

    }
```

}

As you can see, we have an **EntityManagerFactory** as an attribute that is initialized in the post construct method. In the method we created a query and called it the and then the **EntityManager** was closed. In the **save()** method, we began the translation, executed the **persist()** method, and then committed the transaction. Note that when using the Jakarta Persistence without Jakarta Enterprise Bean, we should write more steps to apply the read and write to the database.

Note: This example shows you how to get EntityManager using EntityManager Factory and how the transaction can be scoped using the EntityManager. However, you can use the Jakarta Enterprise Bean that has a good integration with Jakarta Persistence and avoid you writing many of these codes. Our goal in this chapter is not to avoid these codes to show these to you; however, in general, the Jakarta Enterprise Beans is used. *Chapter 5, Jakarta Enterprise Beans* shows you more about this integration.

Now we'll create the **BookBusiness** class. The following is the example:

import net.rhuanrocha.entities.Book;

import

import jakarta.enterprise.context.ApplicationScoped;

```
import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityManagerFactory;

import jakarta.persistence.Persistence;

import java.util.List;

@ApplicationScoped

{

private EntityManagerFactory entityManagerFactory;

@PostConstruct

public void

        entityManagerFactory = Persistence


    }

public List findAll(){
```

```
        EntityManager entityManager =
entityManagerFactory.createEntityManager();

        List books = entityManager

b from Book b join fetch

                .getResultList();

        entityManager.close();

return books;

    }

public Book save (Book book){

        EntityManager entityManager =
entityManagerFactory.createEntityManager();

        entityManager.getTransaction().begin();

//Author needs to be attached to session

!= null){

            Set authors = book
```

```
                    .getAuthors()

                    .stream()


                    .collect(Collectors.toSet());

            book.setAuthors(authors);

        }

        entityManager.persist(book);

        entityManager.getTransaction().commit();

        entityManager.close();

return  book;

    }

}
```

The **BookBusiness** is very similar to **AuthorBusiness** as we'll have a method to find all and to save the books. However, the save method should read the author from the database, because the

Author's objects need to be attached to the session. Now, we'll create the The following is the code of

```
import jakarta.inject.Inject;

import jakarta.ws.rs.GET;

import jakarta.ws.rs.POST;

import jakarta.ws.rs.Path;

import jakarta.ws.rs.Produces;

import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

import java.net.URI;


@Produces(MediaType.APPLICATION_JSON)

@Consumes(MediaType.APPLICATION_JSON)

{

@Inject
```

```
private AuthorBusiness authorBusiness;

@GET

public Response find(){

return Response.ok(authorBusiness.findAll()).build();

    }

@POST

public Response save(Author author){


        authorBusiness.save(author);

return Response


                .build();


    }

}
```

We'll not explain in depth about the **AuthorResource** as it is a Jakarta RESTful resource, and we have a full chapter on it. Just note that we inject the **AuthorBusiness** into it to perform the operations.

Now, we will create the The following is the code of

```
import net.rhuanrocha.business.BookBusiness;
import net.rhuanrocha.entities.Book;

import jakarta.inject.Inject;

import jakarta.ws.rs.GET;

import jakarta.ws.rs.POST;

import jakarta.ws.rs.Path;

import jakarta.ws.rs.Produces;

import jakarta.ws.rs.core.MediaType;

import jakarta.ws.rs.core.Response;

import java.net.URI;


@Produces(MediaType.APPLICATION_JSON)
```

```
@Consumes(MediaType.APPLICATION_JSON)

{

@Inject

private BookBusiness bookBusiness;

@GET

public Response find(){

return Response.ok(bookBusiness.findAll()).build();

    }

@POST


public Response save(Book book){


        bookBusiness.save(book);

return Response
```

```
                .build();

    }

}
```

Now we'll test it by sending a request using To list all the authors, we can use the following command:

curl http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/authors -v

The following is the output:

```
> GET /samplejakartapersistence-1.0-SNAPSHOT/resources/authors HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080

> Accept: */*

>

< HTTP/1.1 200 OK
```

<

<

* *Connection*

[]

To list all the books, we can use the following command:

curl http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/books -v

The following is the output:

> *GET /samplejakartapersistence-1.0-SNAPSHOT/resources/books HTTP/1.1*

> *User-Agent: curl/7.29.0*

> *Host: localhost:8080*

> *Accept: */**

>

< HTTP/1.1 200 OK

<

<

<

<

<

* *Connection*

[]

To add a new author, we can use the following command:

curl -X POST 'http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/authors' -H 'Content-Type: application/json' -d '{"name":"Rhuan Rocha"}' -v

The following is the output:

> POST /samplejakartapersistence-1.0-SNAPSHOT/resources/authors HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080

*> Accept: */**

> Content-Type: application/json

> Content-Length: 22

>

** upload completely sent off: 22 out of 22 bytes*

<

<

<

<

To add a new book, you can use the following command:

curl -X POST 'http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/books' -H 'Content-Type: application/json' -d '{"title":"Jakarta EE Guide","description":"Book about Jakarta EE","authors":[{"id":1}]}' -v

The following is the output:

> POST /samplejakartapersistence-1.0-SNAPSHOT/resources/books HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080

*> Accept: */**

> Content-Type: application/json

> Content-Length: 87

>

** upload completely sent off: 87 out of 87 bytes*

<

<

<

<

Note that you need to add some authors before adding a book, and then can you inform the author in the JSON.

Using JPQL

In the relational database, the SQL is the language used to apply the read and write operations. It is a declarative and powerful language, as you just declare what you want and the mechanism creates the execution plan to deliver it. However, in Java, we work with OOP, and the SQL is not OOP based. Thus, the Jakarta Persistence has the **Jakarta Persistence Query Language** that is a language search data against the persistent entities. The main difference is that the database works with tables, thus the SQL applies searches against tables, and the Jakarta Persistence works with entities. Thus, the SQL is not so good to work inside the Jakarta Persistence as it can generate more complexities. In the preceding example, we saw the **findAll** method that has the JQPL **a from Author** As you can see, you specify the entity to provide an alias to this entity and ask to select the operation you want to return the alias. The following is a generic explanation of the JPQL:

select {alias_entity} from {entity} {alias_entity}

If you want to use the where clause, you can use the following:

select {alias_entity} from {entity} {alias_entity} where {alians_entity}.{property}

Look at the following example:

select a from Author a where a.name=”Rhuan”;

Note that in the JPQL, we navigated throughout the entity as you do when navigating throughout some object in the Java language. It is the main difference between the SQL and JPQL. The same happens when you are navigating through the entities that have a relationship. In the SQL, you should use the **JOIN** to read from multiple tables that have a relationship, and you want to filter it by some parameter of the other table. In the JQPL, you just navigate according to the relationship declared inside the entities. Look at the example as follows:

select b from Book b where b.authors.name=”Rhuan”;

In the preceding example, we are searching for books of the author named Note that it is very easy and closer to the OOP approach.

Parameters

As you can see, it is very easy to be used. Of course, in this example, our scenario is very easy, but even in complex cases, inside the Jakarta Persistence, it tends to be easier than SQL. However, let's imagine we want to pass the value of the search as a parameter. The JPQL permits us to pass the values as parameters, turning the query as more dynamic. To define a parameter in the JPQL, we should use Look at the following example:

select a from Author a where a.name=:name;

To show you an example, we will add a new method to the **AuthorBusiness** to find an author by name. The following is the method called

findByName(String name){

EntityManager entityManager = entityManagerFactory.createEntityManager();

authors = entityManager

a from Author a where

.getResultList();

entityManager.close();

return authors;

}

Note that it is called the **createQuery** method that creates an instance of **Query** class. Also note that we defined the parameter

in the query. Thus, we called the **setParameter** with the same key used in the query (name). To test it, we will modify the **AuthorResource** to receive a request with the name as the query parameter in the URL. Look at the following change done:

```
@GET

    public Response String name){

!= null && !name.isEmpty()){

        }

    }
```

Note that now we have the query parameter If the name is sent in the request, then the **findByName** is called. In other cases, it calls the

To test it, you can use the following **curl** command to add the author named

curl -X POST 'http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/authors' -H 'Content-Type: application/json' -d '{"name":"Rhuan"}' -v

The following is the output:

> POST /samplejakartapersistence-1.0-SNAPSHOT/resources/authors HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080

*> Accept: */**

> Content-Type: application/json

> Content-Length: 16

>

** upload completely sent off: 16 out of 16 bytes*

<

<

<

<

To list the author's named use the following command:

curl 'http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/authors?name=Rhuan' -v

The following is the output:

> GET /samplejakartapersistence-1.0-SNAPSHOT/resources/authors?name=Rhuan HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080

*> Accept: */**

>

< HTTP/1.1 200 OK

<

<

<

* *Connection*

To show an example that we search for data filtering by a property of a relationship, we will modify the In this change, we'll add a new method to find a book of a specific author filtered by the name. Look at the following method **findByAuthorName** that will be added to

```
findByAuthorName(String name){

        EntityManager entityManager =
entityManagerFactory.createEntityManager();

books = entityManager

b from Book b join fetch b.authors a where

                .getResultList();

        entityManager.close();

return books;
```

```
    }
```

Note that we are using the join fetch in the query because we want all the data read together. For the many-to-many relationships, the JPQL uses the LAZY mode to read the data from the relationship. It means that the data is not read until it is needed.

Now, we will modify the method find of the **BookResource** to allow receiving the query parameter named The following is the find method modified:

```
@GET

    public Response String authorName){


!= null && !authorName.isEmpty()){



        }


}
```

To test it, you can use the following **curl** command to add the new

curl -X POST 'http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/authors' -H 'Content-Type: application/json' -d '{"name":"Rhuan"}' -v

The following is the output:

> POST /samplejakartapersistence-1.0-SNAPSHOT/resources/authors HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080

*> Accept: */**

> Content-Type: application/json

> Content-Length: 16

>

** upload completely sent off: 16 out of 16 bytes*

<

<

<

<

To add new use the following command:

curl -X POST 'http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/books' -H 'Content-Type: application/json' -d '{"title":"Jakarta EE Guide","description":"Book about Jakarta EE","authors":[{"id":1}]}' -v

The following is the output:

> POST /samplejakartapersistence-1.0-SNAPSHOT/resources/books HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080

*> Accept: */**

> Content-Type: application/json

> Content-Length: 87

>

** upload completely sent off: 87 out of 87 bytes*

<

<

<

<

To list the books filtered by the author's name, use the following command:

curl 'http://localhost:8080/samplejakartapersistence-1.0-SNAPSHOT/resources/books?authorName=Rhuan' -v

The following is the output:

> GET /samplejakartapersistence-1.0-SNAPSHOT/resources/books?authorName=Rhuan HTTP/1.1

> User-Agent: curl/7.29.0

> Host: localhost:8080

```
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/json
< Content-Length: 111
<
* Connection #0 to host localhost left intact
about Jakarta EE
```

As you can see, using the JPQL, we can create queries with a declarative language that uses the OOP approach, permitting us to create the queries based on the entities and not on the tables as the SQL.

Controlling concurrent access to entity data

In the real world, an application could manage many transactions that operate concurrent writes to the database. Though it serializes the operations and isolates them, you can have a problem with dirty reading. Thus, it can generate problems for the business rule. *Table 6.2* shows you an example of dirty reading as follows:

follows: follows:
follows: follows: follows: follows: follows: follows:
follows:

Table 6.2: *Concurrent Transactions*

As you can see, the rollback backs the data to Thus, the operation in O3 read the João, but the O4 rollbacks the transaction 2 and the author is named Rhuan again. Transaction 1 has the old data generating the dirty reading. Thus, the Jakarta Persistence provides the two ways to working with the concurrent access to the entity's data – these are the optimistic locking and the pessimistic locking. In the following sections, we'll explain more about these lock modes.

Optimistic locking

In the optimistic locking, the entity has an attribute that keeps the entity's version. Thus, for each entity writing operation, the version value of the version is updated. With this, when the data will be updated, the Jakarta Persistence checks the version value, and if the version is different, an exception is thrown, and in case if the version's value is not different, the entity is updated and the version is incremented. The version value is defined annotating the property with Look at the following example:

```
@Entity

{

    ...

@Version

version;

    ...

}
```

Note that the property is of the Long type, but it can be of Integer, long, Long, short, Short, or **java.sql.Timestamp** types.

The optimistic locking is the default lock mode, but you can redefine it by using the following code:

You can also use the following code:

```
= entityManager.createQuery(...);

query.setLockMode(LockModeType.OPTIMISTIC);
```

You have another way to define it, but these are the principal ways.

Thus, the following are the complete codes from the **Author** and **Book** entities:

Author entity:

```
import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Column;

import

@Entity
```

```
{

@Id

@GeneratedValue

id;

@Version

version;

@Column

private String name;

getId() {

return id;

    }

public void id) {

= id;

    }
```

```
getVersion() {

return version;

    }

public void version) {

= version;

    }

public String getName() {

return name;

    }


public void setName(String name) {

= name;

    }

}
```

Book entity:

```
import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Column;

import

import jakarta.persistence.ManyToMany;

import jakarta.persistence.FetchType;

import java.util.Set;

@Entity

{

@Id

@GeneratedValue

private Long id;
```

@Column

private String title;

@Column

private String description;

@ManyToMany

private Set authors;

public Long {

return id;

}

id) {

= id;

}

public String {

```
return title;

    }

title) {

= title;

    }

public String {

return description;

    }

description) {

= description;

    }

public Set {

return authors;

    }
```

authors) {

= authors;

}

}

If you prefer, you can have other kinds of Optimistic lock modes. *Table 6.3* shows these lock modes as follows:

follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

***Table 6.3:** Lock modes*

Pessimistic locking

The optimistic locking is a good approach to work with concurrent access to the entities. However, it should be used when the concurrent access happens a few times. When the optimistic locking is used and the concurrent access happens, an exception is thrown and the process running should read the data again to update or apply a rollback. Thus, if the concurrent write happens

many times, it can degrade the application's feature or generate many rollbacks. Thus, to work in these scenarios, we have pessimistic locking.

In the pessimistic locking, the provider applies a long-term lock on the data until the transaction is completed. Thus, when one transaction applies the long-term lock, another transaction could not read/write the data on the database. To use the pessimistic locking, you can use the following code:

You can also use the following code:

```
= entityManager.createQuery(...);

query.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

The following is the **AuthorBusiness** with changes to work with the concurrent access:

```
import jakarta.annotation.PostConstruct;

import jakarta.enterprise.context.ApplicationScoped;

import jakarta.persistence.EntityManager;

import jakarta.persistence.EntityManagerFactory;
```

```
import jakarta.persistence.LockModeType;

import jakarta.persistence.Persistence;

import net.rhuanrocha.entities.Author;

import java.util.List;

@ApplicationScoped

{

entityManagerFactory;


public void

        entityManagerFactory = Persistence


    }

findAll(){

EntityManager entityManager =
entityManagerFactory.createEntityManager();
```

```
            entityManager.getTransaction().begin();

authors = entityManager

a from Author




                    .getResultList();

            entityManager.getTransaction().commit();

            entityManager.close();

return authors;

      }

name){

EntityManager entityManager =
entityManagerFactory.createEntityManager();

authors = entityManager

a from Author a where
```

```
                    .getResultList();

            entityManager.close();

return  authors;

      }

save  author){

EntityManager  entityManager  =
entityManagerFactory.createEntityManager();

            entityManager.getTransaction().begin();

//If  it  has  an  ID  then  it  is  an  update

!= null){

                  entityManager.lock(author,

            }

            entityManager.persist(author);

            entityManager.getTransaction().commit();
```

```
            entityManager.close();


return  author;

        }

}
```

Note that in the **findAll** method, we are beginning a transaction; besides, it is just read data. It is because to use lock, you should be running inside a transaction scope. Thus, just to show you how you can enable the pessimistic locking, we are starting a transaction. However, in the real world, it does not need a transaction or lock, as at any point, it writes the data.

In the save method, we check if the id is null or not. If the **ID** is null, then it is an insert operation and does not need to lock the entity. But if the **ID** is not null, it is an update operation and will lock the author entity. The lock continues while the process doesn't commit or rollback the transaction.

Creating fetch plans and entity graphs

The Jakarta Persistence defines the entity properties — properties that represent a relationship and its type is another entity — as fetch lazy. It means that these properties are fetched just when needed, that is, when an operation (get) is done to this entity. It is good to improve the performance, as at some point, we'll not need these data, thus these don't need to be fetched. However, at some point, it is better to fetch these data together as an eager fetch to avoid *N+1* Query and some kind of performance issue.

N+1 Query

In the Jakarta Persistence, the *N+1* Query is a well-known scenario where multiple queries are applied to mount the entire entity needed. This name is such because it applies one query to read the main entity, and for each entity returned, a new query is applied to read the relationships. Look at the following example:

Imagine we are querying the **Book** entity, and for each **Book** entity, we need to see the authors. Thus, the first consult retrieves 10 entities, which means, Thus, the number of queries will be 11 as In an overview, it is the *N+1* Query. Thus, we should be able to define a fetch plan to our queries, and change it when needed in runtime.

The *N+1* query is the most common scenario that generates the performance issues to the application. The way to solve it is by reading all the data together in one query.

The decision of which data should be fetched is not so easy in some scenarios, mainly when the entity has many relationships, and the performance is an important criterion to the application. Thus, the entity graphs provide a good way to the fetch plans according to the needs.

The Jakarta Persistence defines the attributes that represent simple columns as and the attributes that represent relationships as However, we can change the default definitions to simple attributes using and to relationship attributes using or The following is an example of it:

```
@Entity

public class Book {

@Id

@GeneratedValue

    private Long id;

@Column
```

```
    private String title;

@Column



    private String description;

= FetchType.EAGER)

    private Set authors;

//getter and setters

}
```

As we are using the Wildfly and its Jakarta Persistence implementation is the Hibernate, we should configure the **hibernate.enhancer.enableLazyInitialization** property. Note that it is not a Jakarta Persistence property, but the Hibernate needs to use the bytecode enhancement to support the lazy loading, that is disabled by default. The following is the **persistence.xml** updated:

```
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
```

>

/>

/>

Using EntityGraph

To use the **EntityGraph** in the Jakarta Persistence, first we need to call the method factory provided by the **EntityManager** and tell the **EntityGraph** what the attributes to be fetched are. Look at the following example:

EntityGraph entityGraph = entityManager

As you can see, we created the **EntityGraph** to the **Book** entity and told to fetch the author's attribute. Thus, we need to pass

the **EntityGraph** to the Query or the The following are both the cases:

Using in a Query:

books = entityManager

b from Book

entityGraph)

.getResultList();

Using in the

properties = new HashMap();

entityGraph);

Note that in both the cases, you should pass the

The following isthe **BookBusiness** class changed to use the

@ApplicationScoped

{

```
        ....

public  List

            EntityManager  entityManager  =
entityManagerFactory.createEntityManager();

            EntityGraph  entityGraph  =



            List  books  =  entityManager



b  from  Book

entityGraph)

                        .getResultList();

            entityManager.close();

return  books;

        }

        ....

}
```

Another good way to use **EntityGraph** is by creating some statics **EntityGraphs** that can be referenced by a name. This feature is called With a we can create static **EntityGraphs** configurations via annotations and in the code. The following is an example:

attributeNodes={

}),

attributeNodes={

})

})

Note that we have the **@NamedEntityGraphs** that has a set of It should be defined in an entity. The following is the **Book** entity:

attributeNodes={

```
        }),
attributeNodes={

        })
})
@Entity
public class Book {
@Id
@GeneratedValue
    private Long id;
@Column
//@Basic(fetch=FetchType.LAZY)
    private String title;
```

```
@Column

//@Basic(fetch=FetchType.LAZY)

    private String description;

= FetchType.LAZY)

    private Set authors;

//getters and setters

}
```

Thus, you should create the **EntityGraph** in your code, described as follows:

```
EntityGraph entityGraph = (EntityGraph)

books = entityManager

b from Book

entityGraph)

                    .getResultList();
```

Note that we called **EntityManager.getEntityGraph** as the **EntityGraph** is already created and we will just use that. It is good as the application does not spend time creating the The following is the **BookBusiness** class:

```
@ApplicationScoped

{

    ...

public List

        EntityManager entityManager =
entityManagerFactory.createEntityManager();

        EntityGraph entityGraph = (EntityGraph)

        List books = entityManager

b from Book

entityGraph)

                .getResultList();

        entityManager.close();
```

```
    return books;


    }

  ...

}
```

As you can see, the **EntityGraph** is an easy way to tell the Jakarta Persistence what data to fetch.

Conclusion

The Jakarta Persistence is a good and important specification of the Jakarta EE, as often the application connects to some relational database and doing so using the JDBC is not an easy way. Throughout the time, the relational database has reduced, but it is an effect of rebalancing of the market caused by NoSQL. However, it continues being used a lot and should continue being used for a long time. Working with the Jakarta Persistence, we are able to apply the operations to the relational database using the OOP perspective.

In the next chapter, we'll talk about the Jakarta messaging, and understand when to use and how to use it.

CHAPTER 7

Jakarta Messaging

Introduction

In this chapter, we will explain what Jakarta Messaging is and what are its aims. Besides, we'll show you how we can create a queue, a topic, a producer, and a consumer, and how we can create the message-driven bean. After studying this chapter, you will be able to use Jakarta Messaging to make the integrations between applications.

Structure

In this chapter, we will learn the following topics:

Understanding the Jakarta Messaging

Creating a producer to queue

Creating a consumer to queue

Creating a producer to topic

Creating a consumer to topic

Creating a consumer using message-driven bean

Objectives

The main objectives of this chapter are to enable the readers to use the Jakarta Messaging in a project, produce messages to queue, consume messages from queue, produce messages to topic, consume messages from topic, and consume messages using the message-driven beans.

Understanding the Jakarta Messaging

When we develop an enterprise application, in general, this application communicates with the other applications. Thus, many a times, the enterprise application needs to send the messages to each other in a loosely coupled and asynchronous way. With this, the enterprise application should have some known way to send the messages between these applications. The applications that will make a communication should know the message pattern and how to send and receive the messages. The pattern is very important to integrate the applications in a good way because if all the applications know the message's patterns, all the applications will know how to send and receive the messages. With this, the Jakarta EE has the Jakarta Messaging to provide a solution based on the specification and to treat the integration between the Jakarta applications using the messaging approach. The specification at https://jakarta.ee/specifications/messaging/3.0/jakarta-messaging-spec-3.0.pdf describes the following aims to Jakarta Messaging:

objectives of Jakarta Messaging are:

to provide Java applications with the messaging functionality needed to implement sophisticated enterprise applications

to define a common set of messaging concepts and facilities

to minimize the concepts a Java language programmer must learn to use enterprise messaging products

to maximize the portability of Java messaging applications between different messaging products."

Jakarta Messaging defines the producer that produces the message and sends it to a queue or topic. It defines the consumer as well, that consumes the message from the queue or topic. Throughout this chapter, we will explain the difference between a queue and a topic, but both work with the **First In First Out** approach. The producer and consumer can be in different applications or in the same application. Although, for the most part, they are in different applications.

The Jakarta Messaging has a good integration with Jakarta Enterprise Bean and Jakarta Transaction API. It means that Jakarta Messaging can participate in the transaction and can take advantage of the container-managed transaction from Jakarta Messaging. It also means that if the Jakarta Messaging is being used inside the session bean of Jakarta Enterprise Bean, the rollback of the sending message can be called in case of any unchecked exception, without the action of the developer. Even if the message was sent and the transaction was not committed, the message can be rolled back.

Queue

The queue approach allows the application to send a message to a queue and the other client gets this message from the queue. It is called a **point-to-point** messaging style as well. In case if a client gets the message from the queue, another client will not be able to get the same message, but will get the next. The queue retains a message until this message is consumed or when it expires. Thus, if we have many clients getting the message from the queue, it does not guarantee which client will get the message. Each message is consumed by only one client. The queue approach should be used when the message should be read and processed by only one client.

Topic

In the topic approach, which is known as the Publish/Subscribe messaging style as well, the message is read and processed by all the clients subscribed to the topic. As the name shows, the topic works with a publisher that sends the message to the topic, and the subscribers read the message from the topic and process it. The main difference between a queue and a topic is that the topic can have multiple consumers and all the consumers subscribed will receive the messages. By default, the consumer just receives the messages sent after subscribing, but Jakarta Messaging provides a way to allow the consumer to consume messages sent to the topic before subscribing to this topic. To do that, use the PERSISTENT delivery mode for the publishers and a durable subscription for the subscribers.

Note that the subscription and the consumer are not the same thing. The topic has the subscription that receives all the messages sent to the topic, according to the configuration, and the consumer that is associated with the subscription. One subscription can have multiple consumers and this is called a **shared** or you can have a subscription with just one consumer, that is called an unshared subscription. Throughout this chapter, we'll explore more of these subscriptions.

In Jakarta Messaging, the messages can be consumed synchronously or asynchronously. In a synchronous way, the consumer calls the receive method to get a message and it'll

block until a message arrives or it takes a time out. In an asynchronous way, a message listener, or a **Message-Driven Bean** is registered to the listener of the messages.

The Jakarta Messaging has two parts – the connection factory and the destinations. The connection factory encapsulates the logic and its configurations to create the connections. The connection factory is an instance of the or **TopicConnectionFactory** interface. The **ConnectionFactory** can be injected using CDI, shown as follows:

```
@Resource(lookup = "java:jboss/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```

The Wildfly provides but you can create new ones.

A destination is an object that represents the target of the message that will be sent. In case of the queue, the object will be of and in case of a topic, the object will be of The destination can be injected using CDI as well, shown as follows:

```
@Resource(lookup = "java:/jms/queue/MyQueue")
private static Queue queue;
@Resource(lookup = "java:/jms/topic/MyTopic")
private static Topic topic;
```

Message

The message is represented by an object that can be of these types – and These messages consist of a header, properties, and a body. The header message contains information about the identity and route, properties are additional fields that can be added to the message, and the body contains the content that the client sent to the provider.

When a client wants to send a message to a queue/topic, it creates a producer and the message. Then the message is sent using the producer. Thus, the client that wants to consume the message should create a consumer and read the message. The message will be of the type described earlier. The messages are read-only, which means it cannot be modified after being sent. The same message can be sent multiple times.

To show a clear example of Jakarta Messaging uses, we'll create a scenario of sending an email. We'll have one producer that will produce the message of an email, and the consumer will get this message and send it to the email. Both the consumer and the producer will be in the same application, but it can be separated as well. Thus, we'll create a new project called **samplejakartamessaging** using the archetype described in *Chapter 1, Introduction to* The following maven command could be used to it:

mvn archetype:generate -DarchetypeGroupId=net.rhuanrocha -DarchetypeArtifactId=jakartaee9-war-archetype -

DarchetypeVersion=1.0.SNAPSHOT -DgroupId=net.rhuanrocha -DartifactId=samplejakartamessaging

Note that you can choose another **-DgroupId** and/or

As we are working with Wildfly, the following commands should be applied to CLI in

```
./jboss-cli.sh --connect
jms-queue add --queue-address=EmailQueue --entries=java:/jms/queue/EmailQueue
jms-topic add --topic-address=EmailTopic --entries=java:/jms/topic/EmailTopic
/subsystem=messaging-activemq/server=default/pooled-connection-factory=activemq-ra:write-attribute(name=client-id,value="myclient")
```

Note that we are adding a queue and a topic that will be used throughout this chapter. In order to use Jakarta Messaging in the Wildfly, you should use the full profile, which means that you should use **standalone-full.xml** or **standalone-full-ha.xml** as a configuration file. Wildfly is not in the scope of this book, but we have some tips in *Chapter 1, Introduction to* and you can check the documentation of Wildfly at **https://docs.wildfly.org** to see how to use the full profile.

Creating a producer to queue

To create the example described earlier, first we'll create a producer to send the message to a queue. Thus, as we want to send an email using Jakarta Messaging, we'll create one class to represent the email, another to represent the endpoint, and one more to be the service. The service will be a Jakarta Enterprise Bean as the Jakarta Messaging resources are eligible to work with the transactions. Thus, the following is the **Email** class that represents the email:

Email implements Serializable {

subject;

to;

message;

getSubject() {

return subject;

}

subject) {

```
= subject;

    }

getTo() {

return to;

    }

to) {

= to;

    }

getMessage() {

return message;

    }

message) {


= message;
```

```
    }

}
```

As the instance of the **Email** class will be sent as a message to the queue, this class should implement Only the classes that implement it can be sent to the queue.

Note that the class has the subject, to, and the message attributes that are important to the consumer that will process it and send it.

Now we'll create the **EmailService** class, which is responsible for putting the message with the email information inside the queue. The **EmailService** class is as follows:

```
import

import jakarta.ejb.Stateless;

import jakarta.jms.JMSConnectionFactory;

import jakarta.jms.JMSContext;

import jakarta.jms.JMSProducer;

import jakarta.jms.Queue;
```

```
import jakarta.jms.JMSException;

import jakarta.jms.MapMessage;

import jakarta.inject.Inject;

import jakarta.jms.*;

import net.rhuanrocha.bean.Email;

@Stateless

{

@Inject


private JMSContext context;


@Resource(mappedName

private Queue queue;

public void sendEmail(Email email){
```

```
        JMSProducer producer = context.createProducer();

        producer.send(queue,email);

    }

}
```

Note that we have **JMSContext** and it is using the **ConnectionFactory** with the JNDI It is injected using CDI. This **ConnectionFactory** is already provided by Wildfly, as follows:

```
@Inject
@JMSConnectionFactory( "java:jboss/DefaultJMSConnectionFactory")
private JMSContext context;
```

The destination is injected using CDI and JNDI as well. The JNDI is **java:/jms/queue/EmailQueue** that was created by us earlier.

```
@Resource(mappedName = "java:/jms/queue/EmailQueue")
private Queue queue;
```

Thus, we create an instance of **JMSProducer** and send the message. Note that the parameters passed to the send method is the queue destination, that is, the queue and the email that will be the message, which is as follows:

```
JMSProducer producer = context.createProducer();
producer.send(queue,email);
```

Now, we will create the endpoint that will receive a request to send the email. In this endpoint, the session bean will be injected using CDI. The following is the **EmailEndpoint** that is a Jakarta RESTful resource:

```
public class EmailEndpoint {

@Inject

    private EmailService emailService;

@POST

    @Consumes(MediaType.APPLICATION_JSON)

    public Response sendEmail(Email email){
```

```
    }

}
```

Now, we want to test this feature. To apply the test, we will send an HTTP request to the endpoint using the following **curl** command:

```
curl -X POST 'http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email' -H 'Content-Type: application/json' -d '{"to":"teste@teste.com","subject":"My Subject","message":"Email test"}' -v
```

The following is the output:

```
> POST /samplejakartamessaging-1.0-SNAPSHOT/resources/email HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
> Content-Length: 70
>
* upload completely sent off: 70 out of 70 bytes
< HTTP/1.1 200 OK
```

< **Connection: keep-alive**
< **Content-Length: 0**

To check if the message was added to the queue, you can execute the following command at the Wildfly CLI:

jms-queue count-messages --queue-address=EmailQueue

It should return a number greater than 0 if you added some message and do not have any consumer.

Now that we created the producer, we want to create the consumer to read the message from the queue.

Creating the consumer to queue

To consume the message, we will create a class called **EmailConsumer** to be the consumer and we will call the consumer by an HTTP request in the **EmailEndpoint** class. As we use the Jakarta Enterprise Beans, the transaction starts automatically, thus, we'll need to pay attention as the consumer is synchronous and keeps waiting for a new message or until a time out. Thus, we have two points here to which we need to pay attention – one is the long-running transaction and the other is the Jakarta Enterprise Bean commit/rollback transaction at the end of the method. If the method execution is blocked, it will not commit or rollback the transaction, and in case of an exception caused by the time out, the rollback will be applied. If it is a bit confusing, don't worry, as we'll explain it more with the following code:

```
import jakarta.annotation.Resource;

import jakarta.ejb.Stateless;

import jakarta.ejb.TransactionAttribute;

import jakarta.ejb.TransactionAttributeType;

import jakarta.inject.Inject;

import jakarta.jms.JMSConnectionFactory;
```

```
import jakarta.jms.JMSConsumer;

import jakarta.jms.JMSContext;

import jakarta.jms.Queue;

import net.rhuanrocha.bean.Email;

import java.util.logging.Logger;

@Stateless

{


private Logger logger =

@Inject


private JMSContext context;

=

private Queue queue;
```

```
        JMSConsumer consumer = context.createConsumer(queue);
```

to read

```
        Email email =

        sendEmail(email);

    }

email){

//Logic to send email




    }

}
```

As you can see, the **consumeMessage** method creates the consumer and calls At this point, the code will wait until the queue has some message or when the call takes a time out. The **sendMail** method is a method that simulates the email sending

logic. Now, we will create a method in the **EmailEndpoint** class to receive an **HTTP GET** request and consume the queue, as follows:

```
public class EmailEndpoint {

@Inject

    private EmailService emailService;

@Inject

    private EmailConsumer emailConsumer;

@POST


    public Response sendEmail(Email email){



    }
```

```
    }

}
```

Now, to test it, we will use the following **curl** command:

```
curl 'http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email' -v
```

The following is the output:

```
> GET /samplejms-1.0-SNAPSHOT/resources/email HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
>

< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 0
```

The information about the message should be logged at the Wildfly console. Note that the consumer is synchronous.

Creating a producer to topic

Creating a producer to the topic is very similar to creating a producer to the queue. The difference is that the destination's subclass is **Topic** and the JNDI should point to a topic inside the Wildfly. Thus, to show it in practice, we'll create a new class called **EmailServiceTopic** that is the class responsible to create the producer and send the message to the topic. The name is just to tell us that it is working with a topic. Look at the following example:

```
import jakarta.annotation.Resource;

import jakarta.ejb.Stateless;

import jakarta.inject.Inject;

import jakarta.jms.JMSConnectionFactory;

import jakarta.jms.JMSContext;

import jakarta.jms.JMSProducer;

import jakarta.jms.Topic;

import jakarta.jms.JMSException;
```

```
import  jakarta.jms.MapMessage;

import  net.rhuanrocha.bean.Email;

@Stateless

{

@Inject


private  JMSContext  context;

=

private  Topic  topic;

email){


            JMSProducer  producer  =  context.createProducer();

            producer.send(topic,email);

      }
```

```
email) throws JMSException {

        JMSProducer producer = context.createProducer();

        MapMessage mapMessage = context.createMapMessage();

email.getSubject());

email.getTo());

email.getMessage());

        producer.send(topic,mapMessage);

    }

}
```

Thus, inside the **EmailEndpoint** class, we will add a new method called **sendEmailTopic** which is responsible for calling as follows:

```
public class EmailEndpoint {

@Inject

    private EmailService emailService;

@Inject

    private EmailServiceTopic emailServiceTopic;

@Inject

    private EmailConsumer emailConsumer;

@POST

    @Consumes(MediaType.APPLICATION_JSON)

    public Response sendEmail(Email email){
```

```
    }

@GET

    public Response startConsumer(){



    }

@POST

    @Consumes(MediaType.APPLICATION_JSON)

    public Response sendEmailTopic(Email email){



    }

}
```

To test it, you can use the following **curl** command:

curl -X POST 'http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email/topic' -H 'Content-Type: application/json' -d '{"to":"teste@teste.com","subject":"My Subject","message":"Email test"}' -v

The following is the output:

```
> POST /samplejakartamessaging-1.0-SNAPSHOT/resources/email/topic HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
> Content-Length: 70
>
* upload completely sent off: 70 out of 70 bytes
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Length: 0
```

To check if the message was added to the topic, you can execute the following command at the Wildfly CLI:

```
jms-topic count-messages --topic-address=EmailTopic
```

It should return a number greater than 0 if you added some message and do not have any consumer.

Now that we created the producer, and now we want to create the consumer to read the message from the topic.

Creating a consumer to topic

The main difference between a topic and a queue is noted from the consumer's perspective. From the producer's perspective, both are very similar, and the developer will use similar codes. As we said earlier, the topic works with a subscription and one subscription can have many consumers. Thus, first the subscription is created and then the consumer is associated with the subscription.

A subscription can be nondurable or durable. In a nondurable subscription, the life scope of the subscription depends on some active consumer. It means that the nondurable subscription will exist while it has some active consumer. In the durable subscription, the subscription exists even if it does not have any active consumer. It means that the consumer, when associated with the durable subscription, will be able to read all the messages produced to the topic while the subscription exists. Furthermore, the subscription can be unshared or shared. In the unshared subscription, only one consumer is allowed to be associated with the subscription, whereas in the shared, the subscription can be associated with multiple consumers. So, now we will create a durable, nondurable, shared, and unshared subscription. Note that our goal here is not about the business logic, but it is possible that you will see some strange things from the business perspective. The goal here is to show you how to create the consumer topic and explore the possibilities of this feature.

Thus, to show you how to create these kinds of subscriptions, we will create another class called **EmailConsumerTopic** which will be one method for each kind of subscription. The first subscription that we will see is the nondurable unshared subscription. Look at the following code:

```
import net.rhuanrocha.bean.Email;

import

import jakarta.ejb.Stateless;

import jakarta.inject.Inject;

import jakarta.jms.JMSConnectionFactory;

import jakarta.jms.JMSConsumer;

import jakarta.jms.JMSContext;

import jakarta.jms.Topic;

import net.rhuanrocha.bean.Email;

@Stateless
```

```
{

private Logger logger =

@Inject


private JMSContext context;

@Resource(mappedName

private Topic topic;

public void consumeMessageNondurableUnshared(){

        JMSConsumer consumer = context.createConsumer(topic);

to read

        Email email =

        sendEmail(email);

    }


}
```

Now we will update the **EmailEndpoint** class to add a new method to request the consumer of the topic, and this new method will be oriented by one query parameter to select the type of subscription we want to request. Look at the following code:

```
{

@Inject

private EmailService emailService;

@Inject

private EmailServiceTopic emailServiceTopic;

@Inject

private EmailConsumer emailConsumer;

@Inject

private EmailConsumerTopic emailConsumerTopic;

@POST

@Consumes(MediaType.APPLICATION_JSON)
```

```
public Response sendEmail(Email email){
        emailService.sendEmail(email);
return Response.ok().build();
    }
@GET
public Response startConsumer(){
        emailConsumer.consumeMessage();
return Response.ok().build();
    }
@GET

public Response String typeSubscription){
        switch (typeSubscription){
```

```
                case

                case

                case

                        emailConsumerTopic.consumeMessageNondurableUns
hared();

            }

return Response.ok().build();

        }

@POST

@Consumes(MediaType.APPLICATION_JSON)

public Response sendEmailTopic(Email email){

            emailServiceTopic.sendEmail(email);
```

```
return Response.ok().build();

    }

}
```

Note that the other cases will be implemented when we add new methods to the other subscription types.

Thus, when we call the **context.createConsumer** method to a topic, the nondurable and unshared subscription will be created. It means that just the message produced before the consumer is created will be read. To test it, we can apply the following **curl** command:

```
curl http://localhost:8080/samplejms-1.0-SNAPSHOT/resources/email/topic?typeSubscription=NONDURABLE_UNSHARED -v
```

Note that, in case of no messages, the request will be waiting until some message is produced, or when the time out happens. Thus, we can execute the following **curl** command to produce the message:

```
curl -X POST 'http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email/topic' -H 'Content-Type:
```

application/json' -d '{"to":"teste@teste.com","subject":"My Subject","message":"Email test"}' -v

Note that if you execute the **curl** commands in another order, the message will not be read as it is nondurable.

Now, we will show you how to consume the topic using a durable and unshared subscription. To do that, we will add a new method called **consumeMessageDurableUnshared** to Look at the following code snippet:

```
public void consumeMessageDurableUnshared(){
JMSConsumer consumer =
context.createDurableConsumer(topic,"Mysubscription1");
logger.info("Trying to read message");
Email email = consumer.receiveBody(Email.class);
sendEmail(email);
}
```

As you can see, now we are calling the **context.createDurableConsumer** method and passing the destination and the subscription name.

Now, we will add the call to this method at the **EmailEndpoint** class. Look at the following code snippet:

public Response String typeSubscription){

```
        switch (typeSubscription){

            case

                emailConsumerTopic.consumeMessageDurableUnshare
d();


            case


            case


                emailConsumerTopic.consumeMessageNondurableUns
hared();

        }

return Response.ok().build();

    }
```

To test it, we can apply the following **curl** command:

curl http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email/topic?typeSubscription=DURABLE_UNSHARED -v

Note that, in case of no messages, the request will be waiting until some message is produced, or when the time out happens. Thus, we can execute the following **curl** command to produce the message:

curl -X POST 'http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email/topic' -H 'Content-Type: application/json' -d '{"to":"teste@teste.com","subject":"My Subject","message":"Email test"}' -v

Now, we will add the durable shared subscription example by adding a method called **consumeMessageDurableShared** to the Look at the following code snippet:

JMSConsumer consumer1 =

JMSConsumer consumer2 =

to read message from

Email email1 =

sendEmail(email1);

to read message from

Email email2 =

sendEmail(email2);

}

As you can see, I'm creating two consumers to the same subscription called It permits us to understand how the shared feature works.

Now, we will add a call to this method at the **EmailEndpoint** class. Look at the following code snippet:

public Response String typeSubscription){

switch (typeSubscription){

case

emailConsumerTopic.consumeMessageDurableUnshare
d();

```
                case

emailConsumerTopic.consumeMessageDurableShared();

                case

                        emailConsumerTopic.consumeMessageNondurableUns
hared();

            }

return  Response.ok().build();

        }
```

To test it, we can apply the following **curl** command:

```
curl  http://localhost:8080/samplejakartamessaging-1.0-
SNAPSHOT/resources/email/topic?
typeSubscription=DURABLE_SHARED  -v
```

Note that, in case of no messages, the request will be waiting until some message is produced, or when the time out happens.

Thus, we can execute the following **curl** command to produce the message:

curl -X POST 'http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email/topic' -H 'Content-Type: application/json' -d '{"to":"teste@teste.com","subject":"My Subject","message":"Email test"}' -v

The final kind of subscription will be the nondurable shared subscription, and to do it, we'll create a new method called **consumeMessageNondurableShared** inside the **EmailConsumerTopic** class. The following is the code snippet to this:

JMSConsumer consumer1 =

JMSConsumer consumer2 =

to read message from

Email email1 =

sendEmail(email1);

to read message from

Email email2 =

sendEmail(email2);

```
}
```

Now, we will add the call to this method at the **EmailEndpoint** class. Look at the following code snippet:

```
public Response String typeSubscription){

        switch (typeSubscription){

            case

                emailConsumerTopic.consumeMessageDurableUnshare
d();

            case

emailConsumerTopic.consumeMessageDurableShared();

            case

emailConsumerTopic.consumeMessageNondurableShared();
```

```
                    emailConsumerTopic.consumeMessageNondurableUns
hared();

            }

return  Response.ok().build();

      }
```

To test it, we can apply the following **curl** command:

```
curl http://localhost:8080/samplejakartamessaging-1.0-
SNAPSHOT/resources/email/topic?
typeSubscription=NONDURABLE_SHARED -v
```

Note that, in case of no messages, the request will be waiting until some message is produced, or when the time out happens. Thus, we can execute the following **curl** command to produce the message:

```
curl -X POST 'http://localhost:8080/samplejakartamessaging-1.0-
SNAPSHOT/resources/email/topic' -H 'Content-Type:
application/json' -d '{"to":"teste@teste.com","subject":"My
Subject","message":"Email test"}' -v
```

Note that all of these calls are synchronous and blocking. The Jakarta EE has a feature to create an asynchronous consumer by the integration of the Jakarta Enterprise Bean with the Jakarta Messaging. This is called **Message-Driven Bean** which was already shown in *Chapter 5, Jakarta Enterprise* We will be talking more about it here, but there will be more details about the Jakarta Messaging configurations.

Creating a consumer using message-driven bean

In this section, we'll create two MDBs, one to the queue and the other to the topic. To the Queue, it will be very similar to the MBD created in *Chapter 5, Jakarta Enterprise*

As you have seen, all the code we wrote were using a synchronous way, which means the call to consume the queue is blocking. Thus, in our test, we sent a **POST** request to add the message to the queue, and a **GET** request to read the message from the queue. Now, the first difference that you will see is to test the code, we will just need to send a **POST** request because the code to consume the message will already be listening to the queue and automatically be read it.

To create the MDB, we need to create a class that extends **jakarta.jms.MessageListener** and annotate this class with **@MessageDriven** to configure the properties needed. To do that, we will create a class called **EmailMdb** which will be the MDB to read the message and send an email. Look at the following code:

```
import jakarta.ejb.ActivationConfigProperty;

import jakarta.ejb.MessageDriven;

import jakarta.jms.JMSException;
```

```
import  jakarta.jms.Message;

import  jakarta.jms.MessageListener;

import  net.rhuanrocha.bean.Email;

import  java.util.logging.Logger;

= {

=

                        propertyValue  =


=

                        propertyValue  =

})

{

private  Logger  logger  =

@Override

message)  {
```

```
to read message from

try {

        Email email =

        sendEmail(email);

    } catch (JMSException e) {

        logger.severe(e.getMessage());

    }

}

email){

//Logic to send email




}

}
```

As you can see, we annotated it with **@MessageDriven** and passed the configurations using the as follows:

= {

=

propertyValue =

=

propertyValue =

})

When the application starts, the MDB begins to listen to the queue. The queue should already exist and be accessible to the application. Now we can test it to send a message using the following **curl** command:

curl -X POST 'http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email' -H 'Content-Type: application/json' -d '{"to":"teste@teste.com","subject":"My Subject","message":"Email test"}' -v

You will see that the message is printed to the log because the listener will consume the message posted to the queue.

Listening topic

As you know, consuming a topic is a bit different, because it has a subscription that can be shared or unshared, durable or nondurable, and it has the consumers related to the subscription. Thus, we will see in practice how to create an MDB and to consume the topics for each kind of subscription. Look at the following code that creates an MDB to consume the topic with unshared nondurable subscription:

```
import jakarta.ejb.ActivationConfigProperty;

import jakarta.ejb.MessageDriven;

import jakarta.jms.JMSException;

import jakarta.jms.Message;

import jakarta.jms.MessageListener;

import net.rhuanrocha.bean.Email;

import java.util.logging.Logger;

= {
```

=

propertyValue =

=

propertyValue =

propertyValue =

=

propertyValue =

=

})

{

private Logger logger =

@Override

```
message) {

to read message from

try {

                Email email =

                sendEmail(email);

        } catch (JMSException e) {

                logger.severe(e.getMessage());

        }

    }

email){

//Logic to send email




    }
```

}

As you can see, the differences are the properties in the annotation. The following is the configuration to the unshared nondurable subscription:

= {

=

propertyValue =

=

propertyValue =

propertyValue =

=

propertyValue =

=

})

To use the durable subscription, you should configure the **subscriptionDurability** with Look at the following example:

= *{*

=

propertyValue =

=

propertyValue =

propertyValue =

=

propertyValue =

=

})

Finally, to configure a shared subscription, you should configure the **shareSubscription** with true. Look at the following configuration:

= {

=

propertyValue =

=

propertyValue =

propertyValue =

=

propertyValue =

=

})

When the application starts, the MDB begins to listen to the topic. The topic should already exist and be accessible to the application. Now we can test it to send a message using the following **curl** command:

curl -X POST 'http://localhost:8080/samplejakartamessaging-1.0-SNAPSHOT/resources/email/topic' -H 'Content-Type: application/json' -d '{"to":"teste@teste.com","subject":"My Subject","message":"Email test"}' -v

You will see that the message is printed to the log because the listener will consume the message posted to the topic.

Conclusion

The Jakarta Messaging specification is a very important specification, mainly in this era as we see the Cloud Native Application concept grow up rapidly. The Jakarta Messaging permits a low coupling between the applications, as it promotes an asynchronous communication and the consumer should not know about the producer and the producer should not know about the consumer. The consumer and the producer follow a logical contract regarding the data and it is just what is needed. In this chapter, we understood the basics of the Jakarta Messaging, and you are able to use this specification in a Jakarta Application.

In the next chapter, we will talk about Jakarta Security and how to use it.

CHAPTER 8

Jakarta Security

Introduction

In this chapter, we will explain what Jakarta Security is and what its aims are. Besides, we'll show you how we can secure a web application, how we can implement a user's authentication programmatically, how we can implement a custom identity store, and how we can implement a custom database identity store.

Structure

In this chapter, we will discuss the following topics:

Understanding the Jakarta Security

Securing a web application

Implementing a user's authentication programmatically.

Securing a Jakarta Enterprise Bean

Implementing a custom identity store

Objectives

The main objectives of this chapter are to enable the readers to use the Jakarta Security in a better way, secure a web application, implement a user's authentication programmatically, secure a Jakarta Enterprise Bean, and implement a custom identity store.

Understanding Jakarta Security

Security is a wide concept and can appear with different approaches according to the context it is being used in. Security can tell us about the network's security, user's security, OS's security, and so on. All of these kinds of security can impact the business rule, but the scope of the Jakarta Security specification does not provide a solution to all these approaches. Actually, the scope of this specification is to provide a user's security mechanism managed by the Jakarta context. It is very important to understand as the name can lead the developer to think that all the security approaches are inside the Jakarta Security. This raises the question, what does the user need as security in general?

As we have said earlier, the Jakarta EE specifications provide a pattern solution to the enterprise problems. It means that the specification should provide a solution to the common problems. If it has a rare problem, the specification doesn't need to care about it. With this, inside the user's authentication scenario, the most common problems are about authentication and authorization.

The Jakarta Security specification provides a set of solutions to provide authentication and authorization solutions managed by the Jakarta context. It means that the Jakarta EE has many abstractions to facilitate the developer to make authentication and

authorization of the users. This specification provides a plug-in interface for the authentication and identity store — which has information about the user, that is used to validate the user — and can be used with annotation or programmatically. The Jakarta Security has four main features provided by the API, which are as follows:

HTTP authentication mechanism

Identify store

Credential

Security context

HTTP authentication mechanism

Modern applications provide a communication interface using the HTTP protocol. In the past, it was common to have the Java applications using the EJB clients or CORBA to make the communications without using the HTTP protocol. However, today it is very different, and the HTTP is the protocol that is widely used. With this, Jakarta Security brings the HTTP authentication mechanism, to provide a solution to the applications that need to use the authentication using the HTTP protocol.

The Jakarta Security provides the **HttpAuthenticationMechanism** interface. It works as a contract and the application can implement its authentication mechanism using this interface. Furthermore, Jakarta Security already provides the implementations to three mechanism types, which are – basic HTTP authentication, form-based authentication, and a customized form-based authentication. These implementations can be used by using the annotation and **@CustomFormAuthenrtication MechanismDefinition** respectively. We'll read more about it in this section.

These annotations are used to configure the authentication mechanism to the application. Thus, we just need to annotate a CDI bean with one of these annotations and the application will be configured. The implementation of these mechanisms is a built-in bean that has the scope Thus, it can be configured in a CDI bean, but I recommend you create a CDI bean responsible

for just configuring the Jakarta Security to the application, and put this CDI bean with the scope Look at the following example:

```
        loginToContinue =

                loginPage =

                errorPage =

                useForwardToLogin = true

        )

)

@ApplicationScoped

public class AuthenticationConfig {

}
```

As you can see, the preceding example is configuring the form-based authentication mechanism.

The configuration of the other authentication mechanisms works in the same way. We'll see it soon.

Identify_store

To validate the authentication, we need to have a set of information about the identity that is known by the application. With this, the system reads the input from request and checks if this information matches with the information contained in the set of information.

These sets can be kept in a database, file system, or memory (used just to test the scenarios).

To provide an abstraction to it and to represent this set of information, we have the Identity Store, which as the name suggests, represents a store of identities, group membership information, and information sufficient to allow it to validate.

The Jakarta Security provides the **IdentifyStore** interface to represent it. This interface can be implemented according to the store you want to use. The following is the interface provided by Jakarta Security:

{

enum ValidationType { VALIDATE, PROVIDE_GROUPS }

CredentialValidationResult credential);

Set validationResult);

Set

}

By default, the Jakarta Security provides the implementation to LDAP and Database, but you can create a custom implementation, if needed. To use these implementations, you can use **@LdapIdentityStoreDefinition** or **@DatabaseIdentityStoreDefinition** annotation, respectively. The following is the example:

dataSourceLookup =

callerQuery = "select password from users where username =

groupsQuery = "select groupname from groups where username =

@ApplicationScoped

public class AuthenticationConfig {

```
}
```

As you can see, we used the Database Identify Store to just annotate the CDI bean.

Credential

To authenticate a user to the application, the user needs to inform the credential information. It can be a token or some user information like the username and password. Jakarta Security provides an interface called *Credential* to represent the credential information. Each implementation of this interface represents a kind of credential. The **IdentiyStore** could work with any credential implementation.

The credential's information represents that the credential implementation is used by **IdentifyStore** to validate whether the user can be authenticated or not.

Context security

As you could see, all the security configuration was shown based on the annotation, in a declarative way. However, sometimes we need to do something that is not provided by the annotation approach. That is, sometimes, we need a programmatic approach to reach the business aims. Jakarta Security provides a way for the developers to access the context security and provide methods to help the developer be able to apply the validations and actions that are needed by the business rules.

To do this, Jakarta Security provides the **SecurityContext** interface that can be injected by some CDI bean, as shown in the following code snippet:

```
@Inject

SecurityContext securityContext;
```

We will see more about the programmatic approach throughout this chapter.

To show a clear example of Jakarta Security, we'll create a new project called **samplejakartasecurity** using the archetype described in *Chapter 1, Introduction to* The following maven command could be used to it:

mvn archetype:generate -DarchetypeGroupId=net.rhuanrocha -DarchetypeArtifactId=jakartaee9-war-archetype -DarchetypeVersion=1.0.SNAPSHOT -DgroupId=net.rhuanrocha -DartifactId=samplejakartasecurity

Note: You can choose another -DgroupId and/or -DartifactId.

All the examples of this chapter will be created inside this project.

Securing a Web application

In our example, we'll create a simple Servlet called HelloWorld and we will configure it to ask for authentication from the end-user. Thus, to access this Servlet, the end-user needs to be authenticated. We'll use **FormAuthenticationMechanism** as the authentication mechanism. Furthermore, we'll use **DatabaseIdentityStore** pointing out to the default data source provided by Wildfly. If you prefer, you can create a new data source and use it instead of the default data source.

As in this example, we are using the default data source and it used an in-memory data source; so, we will need to create a starter to populate the database at the startup time. Note that it is not part of our subject in this chapter; however, I'll put it here to facilitate you to apply the tests described. Thus, we have the User and Group entities, and the which is a Singleton and we will start at the startup time. Look at the following example:

User

@Entity

=

public class User {

@GeneratedValue

@Id

private Long id;

=

private String userName;

=

private String password;

// Getters and Setters

}

Group

@Entity

=

public class Group {

@Id

```
@GeneratedValue

    private Long id;

=

    private String userName;

=

    private String groupName;

// Getters and Setters

}
```

DatabaseStarter

```
@Singleton

@Startup

{

@PersistenceContext

private EntityManager entityManager;
```

```
@Inject

private Pbkdf2PasswordHash passwordHash;

@PostConstruct

        User user = new User();



        entityManager.persist(user);

        Group group = new Group();



        entityManager.persist(group);

    }

}
```

Note: We created a user called admin with password admin123, **inside the group/role admin.**

Now, as the first step to configure the Jakarta Security, we'll create a CDI bean as an **ApplicationScoped** and make the configurations about the authentication mechanism and the identity store. The following is the **AuthenticationConfig** class as an example:

```
        loginToContinue = @LoginToContinue(

                loginPage =

                errorPage =

                useForwardToLogin = true

        )

)

@DatabaseIdentityStoreDefinition(

        dataSourceLookup =
```

```
        callerQuery = "select password from users where username =

        groupsQuery = "select groupname from groups where
username =



@ApplicationScoped

public class AuthenticationConfig {

}
```

As you can see, the **@FormAuthenticationMechanismDefinition** defines the login page and error page. If the user is not authenticated, then the user is redirected to the login page. In case of an error regarding authentication, the user is redirected to the error page.

The **@DatabaseIdentityStoreDefinition** uses Wildfly's default data source The **callerQuery** is selected to read the user's password and the **groupsQuery** is selected to read the user's groups/roles.

Now, we can configure it in our endpoint. In this example, we will configure it in a Servlet called HelloWorld. We need to define what the roles that can access the resource are, as shown in the following code snippet:

```
@DeclareRoles({ })

=

public class HelloWorld extends HttpServlet {

@Override

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {


    }

}
```

Note that the roles of this application are admin and customer, but just the admin can access the Servlet.

If you want a more specific configuration, like securing just the **POST** method, you can configure the **httpMethodConstraint** inside the Look at the following example:

"customer" })

=

httpMethodConstraints = {

value =

rolesAllowed =

value =

rolesAllowed =

})

public class HelloWorld extends HttpServlet {

....

}

Note that the **GET** method can be accessed by the customer and the **POST** method by the admin.

To test, you can access the following URL:

http://localhost:8080/samplejakartasecurity-1.0-SNAPSHOT/servlet/hello

Securing Jakarta RESTful resource

As we have seen earlier, the Servlet is not the unique resource to treat the HTTP request/response, but it has the Jakarta RESTful that works with the HTTP protocol as well. Thus, it should provide a way to secure it as well. To secure a Jakarta RESTful resource, we can use the and

Now, we'll create a Jakarta RESTful resource called **HelloWorldResource** that shows the return just to the allowed users. Look at the following code snippet:

import

import

import jakarta.ws.rs.GET;

import jakarta.ws.rs.Path;

})

{

```
@GET

public String hello(){



    }

}
```

Note that just the users with the admin role can access it. If we prefer, we can add the **@RoleAllowed** to the method. The security at the method level has priority over the security configured at the class level. The following code snippet uses the method level approach:

```
import



import

import jakarta.ws.rs.GET;

import jakarta.ws.rs.Path;



})
```

```
{

@GET


public String hello(){


    }

}
```

Note that we have declared all the roles of the application in both.

As we are using the Wildfly, we should allow the role-based security via however, it is not a specification characteristic, but it is a characteristic from Resteasy that is the implementation of Jakarta RESTful specification. The following code snippet shows the

Implementing a user's authentication programmatically

Configuring the authentication using the annotations is a good approach that makes the process easier and faster, avoiding a lot of code. However, sometimes the business rule has a complex rule regarding authentication that cannot be expressed by just using annotation. In general, annotations are a good approach when the authentication rule does not change according to some logic. However, in some cases, we need an authentication rule that has a dynamic behavior, according to some logic.

To show an example of authentication validation programmatically, we'll create a class called which is a Jakarta RESTful resource that returns a hello world message if the user is authenticated, and he/she has the authorization to access it. Look at the following example:

```
import jakarta.inject.Inject;

import jakarta.security.enterprise.SecurityContext;

import jakarta.ws.rs.ForbiddenException;

import jakarta.ws.rs.GET;

import jakarta.ws.rs.Path;
```

```
import  java.security.Principal;


{


securityContext;


hello(){


Principal  principal  =  securityContext.getCallerPrincipal();


throw  new  not

            }

World

      }

}
```

As you can see, we injected the **SecurityContext** using CDI, and then we called the **SecurityContext.isCallerInRole** method to check if the user has the admin role.

The **SecurityContext** has the **getCallerPrincipal** method that can be called to get the Principal information and the **isCallerInRole** method to validate a role. You can also create your custom logic to validate the user. We recommend that you use it when required, as it mixes the task logic with the authentication logic. Using the annotation, you will write a cleaner code.

Using custom form-based HTTP authentication

In some cases, you may need a customs form to make the authentication. Jakarta Security provides a manner to create it, and an annotation called **@CustomFormAuthenticationMechanismDefinition** to tell the Jakarta Security that we want to use a custom form-based HTTP authentication. Thus, we can create our method to validate and write an authentication rule according to the business rule.

To create a custom form-based HTTP authentication, we first need to create an implementation of **HttpAuthenticationMechanism** as shown in the following code snippet:

```
import jakarta.enterprise.context.ApplicationScoped;

import jakarta.security.enterprise.AuthenticationException;

import jakarta.security.enterprise.AuthenticationStatus;

import
jakarta.security.enterprise.authentication.mechanism.http.HttpAuthenticationMechanism;

import
jakarta.security.enterprise.authentication.mechanism.http.HttpMessageContext;
```

```
import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import java.util.LinkedHashSet;

import java.util.Set;

@ApplicationScoped

{



httpServletRequest, HttpServletResponse httpServletResponse,
HttpMessageContext httpMessageContext) {

String username =

String password =

&&

roles = new
```

```
return httpMessageContext.notifyContainerAboutLogin(username, roles);

            }

return httpMessageContext.responseUnauthorized();

      }

}
```

As you can see, we have implemented the **validateRequest** method that has the logic to validate the authentication.

Now, we want to create the CDI bean that has the logic to make the login. Look at the following code snippet:

```
import jakarta.enterprise.context.RequestScoped;

import jakarta.faces.context.FacesContext;

import jakarta.inject.Inject;

import jakarta.inject.Named;

import jakarta.security.enterprise.AuthenticationStatus;

import jakarta.security.enterprise.SecurityContext;
```

```
import
jakarta.security.enterprise.authentication.mechanism.http.AuthenticationPa
rameters;



import  jakarta.security.enterprise.credential.UsernamePasswordCredential;

import  jakarta.servlet.http.HttpServletRequest;

import  jakarta.servlet.http.HttpServletResponse;

import  jakarta.validation.constraints.NotNull;

@Named

@RequestScoped

{

@NotNull

private  String  username;

@NotNull

private  String  password;

@Inject
```

```
private SecurityContext securityContext;

@Inject

private FacesContext facesContext;



== null || password ==



        }

        HttpServletRequest request = (HttpServletRequest)
facesContext.getExternalContext().getRequest();

        HttpServletResponse response = (HttpServletResponse)
facesContext.getExternalContext().getResponse();

        AuthenticationStatus authenticationStatus = securityContext

                .authenticate(request, response,



                        AuthenticationParameters.withParams()

UsernamePasswordCredential(username, password))
```

```
                    );

    }

}
```

As you can see, the login method receives the username and password from the view and calls the **securityContext.authenticate** method to authenticate the user.

Now, we should create a new form that will make the login. In this step, we'll use the JSF, but it is not in the scope of this book. So, if you want to know more about JSF, I suggest you find a book focused on it. The following code snippet shows the which is the form:

```
version="1.0"  encoding="UTF-8"?>
```

/>

/>

/>

Now, we'll annotate the **AuthenticationConfig** with **@CustomFormAuthen ticationMechanismDefinition** and the custom form-based authentication will be used, as follows:

```
    loginToContinue = @LoginToContinue(
        loginPage =
        errorPage =
        useForwardToLogin = true
))
@ApplicationScoped
public class AuthenticationConfig {
}
```

To test, you can access the following URL:

http://localhost:8080/samplejakartasecurity-1.0-SNAPSHOT/resources/hello

Securing a Jakarta Enterprise Bean

As we saw in *Chapter 5, Jakarta Enterprise* the Jakarta Enterprise Bean is used to wrap the business logic. It has many interesting mechanisms to the business logic, such as transaction management, timers, and others. In the real world, it is common for a business logic to be secured to avoid an unauthorized execution. Jakarta Security can be used with Jakarta Enterprise Bean as well, and it has some peculiarities that will be shown throughout this chapter.

To show an example of Jakarta Security with the Jakarta Enterprise Bean, we'll create a **HelloWorldService** that returns a hello world message. It will be a standalone bean and will be secured to permit the users with the admin roles to access it. Thus, as the first step, we'll create a stateless bean as follows:

```
import

import jakarta.ejb.Stateless;

@Stateless

{
```

```
public String sayHello(){

    }

}
```

As you can see, the method **sayHello** returns the message. Only the admin role can access this method.

Now, we'll update the **HelloWorldResource** to add a new method to call the The following is the method that we'll create:

```
@GET


@PermitAll

    public String helloEjb(){


    }
```

Note that we will configure the **@PermitAll** to avoid the authentication validation at the as we want to validate it only on the The following is the code snippet showing the complete

```
import

import

import

import jakarta.inject.Inject;

import jakarta.ws.rs.GET;

import jakarta.ws.rs.Path;

import net.rhuanrocha.ejb.HelloWorldService;


})


{

@Inject

private HelloWorldService helloWorldService;

@GET
```

```
public String hello(){

    }

@GET



@PermitAll

public String helloEjb(){

return helloWorldService.sayHello();

    }

}
```

The following URL can be used to test it:

http://localhost:8080/samplejakartasecurity-1.0-SNAPSHOT/resources/hello/ejb

As we saw earlier, the Jakarta Enterprise Bean has the timer feature. It means that the first part of the task can be done without a user's request and user authentication. We can configure the method to runs as some role. Thus, if some timer executes

the we can configure it to run as the admin. To do this, we can use the **@RunAs** annotation as shown in the following code snippet:

```
    ...

return  helloWorldService.sayHello();

    }

    ...
```

Thus, the method **timer** will be executed as **admin** and the **helloWorldService.sayHello** will not be unauthorized.

Implementing a Custom Identity Store

We saw that the Identity Store is used to validate the credentials and retrieve the groups, also called However, sometimes we need to create a custom Identity Store, as the Identity Stories provided by Jakarta EE does no solve the proposed problem. To do it, the Jakarta EE permits you to create an implementation to **IdentityStore** interface, write the credential's validation logic, and the logic to get the roles. To show you an example of it, we will create an Identity Story that reads the user information from memory. This means that we will create a map in the memory and the credentials and roles will be placed inside this map.

In our example, we'll create two classes, one called which is a representation of the user information, and the other called which is the custom Identity Store. Look at the following code snippet:

User

User {

username;

password;

private roles;

getUsername() {

return username;

}

username) {

= username;

}

getPassword() {

return password;

}

password) {

= password;

}

public getRoles() {

```
return roles;

    }

roles) {

= roles;

    }

User of username, String password, roles){

        User user = new User();

        user.setUsername(username);

        user.setPassword(password);

        user.setRoles(Arrays.asList(roles));

return user;

    }

}
```

Note that we have the role as List, as one user can have many roles, and we have a method factory to create the **User** instance. The following is the **InMemoryIdentityStore** that uses the **User** class to represent the user **class:**

```
import jakarta.annotation.PostConstruct;

import jakarta.enterprise.context.ApplicationScoped;

import jakarta.security.enterprise.credential.UsernamePasswordCredential;

import jakarta.security.enterprise.identitystore.CredentialValidationResult;



import jakarta.security.enterprise.identitystore.IdentityStore;

import java.util.HashMap;

import java.util.HashSet;

import java.util.Map;

import java.util.Set;

@ApplicationScoped

{
```

```
@PostConstruct

        User user =

    }

// init from a file or harcoded

private MapUser> credentials = new HashMap<>();

@Override

{

    }

public CredentialValidationResult

            UsernamePasswordCredential credential) {

        User user = credentials.get(credential.getCaller());

if (credential.compareTo(user.getUsername(), user.getPassword())) {
```

```
CredentialValidationResult(user.getUsername());

            }

return  CredentialValidationResult.INVALID_RESULT;

      }

@Override

public  Set  validationResult)  {

            User  user  =  credentials.get(

                        validationResult.getCallerPrincipal().getName());

HashSet<>(user.getRoles());

      }

}
```

As you can see, this class has a method called which returns the number. It is used to define the sequence of execution of the Identity Stores. Note that all the Identity Stores will be executed, and the Identity Store with minor priority will be executed first. Look at the following ode snippet:

```
@Override

{



    }
```

Furthermore, we have the method called that has the logic to validate the credentials, and the method called **getCallerGroups** that retrieves the roles. Look at the following code snippet:

```
public  CredentialValidationResult

                    UsernamePasswordCredential  credential)  {

            User  user  =  credentials.get(credential.getCaller());

if  (credential.compareTo(user.getUsername(),  user.getPassword()))  {

CredentialValidationResult(user.getUsername());

            }

return  CredentialValidationResult.INVALID_RESULT;

    }
```

```
public Set validationResult) {

        User user = credentials.get(

                validationResult.getCallerPrincipal().getName());

HashSet<>(user.getRoles());

    }
```

To test, you can access the login form at the following URL:

http://localhost:8080/samplejakartasecurity-1.0-SNAPSHOT/login.xhtml

Now, you can also access the following URL:

http://localhost:8080/samplejakartasecurity-1.0-SNAPSHOT/resources/hello

After logging into the application, the return should be the **hello world** message.

Conclusion

The Jakarta Security is a very important specification to the enterprise environment, as the enterprise has very sensitive data flowing throughout the applications and system. Implementing the security is not an easy task. Jakarta Security abstracts many of these concepts and provides tested and well-known solutions. Throughout this chapter, you learned how to use this specification in the best way.

In the next chapter, you will learn how to use the Jakarta Bean Validation and how we can validate the data inputs using this specification.

CHAPTER 9

Jakarta Bean Validation

Introduction

In this chapter, we will explain what Jakarta Bean Validation is and what its aims are. Besides, we will show you how we can validate an entity, how we can use one in a RESTful resource to validate the data coming from the requests, how we can validate the constructors and methods, how we can create a custom constraint, and how we can customize the validator messages.

Structure

In this chapter, we will discuss the following topics:

Understanding the Jakarta Bean Validation

Validating an entity using bean validation

Using bean validation in a RESTful resource

Validating persistence

Creating a custom constraint

Customizing validator messages

Objectives

The main objectives of this chapter are to enable the readers to use the Jakarta Bean in a better way to validate the input data, validate the data coming into the RESTful resources, validate the data in the constructors and methods, create a custom constraint, and customize the validator's messages.

Understanding the Jakarta Bean Validation

Throughout this book, we have shown you many projects and focused on some features from the chapter, but you can ask us about the inputs from these applications. What if the user makes a wrong input to these applications?

In the real world, the applications need to predict the scenarios of a wrong input — like an empty input or invalid data — and provide the treatment to these cases. In general, it is simple to treat, however, it can ruin the business codes with many codes that are not necessarily from the business logic. Depending upon the validation, the code can be very large. One example is the validation regarding the CPF number — number of a Brazilian document — that calculates the 11's module to tell if it is a valid number.

The Jakarta Bean Validation permits us to define the constraints to the objects in a declarative way, by annotations. It means that you can define the constraints to the objects and their properties and methods just by annotating the method with an annotation that represents the constraint you want. The code to validate the input can be abstracted from the business codes. Furthermore, the code to validate can be reused without a problem in many objects. You can call the validation programmatically as well, without using an annotation. Look at the following example:

```
{
```

```
    private String email;

    ...

}
```

As you can see, we have the class **User** that has the property called This property should not be null and should be a valid email. Note that it is clean and easy to define these constraints. Without Jakarta Bean Validation, you will probably create a large code to validate the email, as it has many details that should be validated.

Validating method

The Jakarta Bean Validation permits apply the validation methods and constructors, and it has two types of validation to methods and constructors, which are **preconditions** and In the precondition validation, the validation occurs before the method or constructor is executed. In this, the validation happens to the method's parameters. Look at the following example:

```
{

    ...

String email){

//codes

            ...

    }

}
```

In the preceding code, when the method **setEmail** is called by the frameworks like Jakarta Persistence, Jakarta Faces, and Jakarta

RESTful Web Services, the email parameter is validated if it is not null and if it is a valid email.

In the postcondition validation, the validation occurs after the method execution, and the returning value is validated. Look at the following example:

```
{

    ...

@NotNull

public List< @NotNull Address> getAddresses(){

//codes

            ...

    }


}
```

In the preceding code, the list of addresses should not be null, nor should it contain null elements. Note that the code is more clean and easy to understand.

Validation annotations

The specification defines some built-in constraints that can be used inside the application. The following describes some well-used constraints found in the **jakarta.validation.constraints** package:

The element should be null.

The element should not be null.

The element should be higher or equal to the specified value.

The element should be lower or equal to the specified value.

The element should be a negative value.

The element should be a positive value.

The element should have a size between the specified boundaries.

The element should not be null and should contain at least one non-whitespace character.

The element should not be null and should not be empty.

The element should be in a valid format of the email address.

You can create new constraints if needed. We will see more of it throughout this chapter.

Using bean validation in a RESTful resource

It is very common that the RESTful resource applies some validations about the data and returns a 400 error (Bad Request) when the data is not as expected. The Jakarta Bean Validation has good integration with the Jakarta RESTful. It means that if the data is not according to the constraints configured using the Jakarta Bean Validation, the Jakarta RESTful resource returns the Bad Request error.

To show how to use the Jakarta Bean Validation with a RESTful resource, we will create a scenario where the RESTful resource receives a user JSON, and it needs to check if the name and email were informed (is not blank) and if the email is valid according to the email pattern. So, we'll create two classes, one to represent the user information and the other to be the RESTful resource. Look at the following code:

```
import jakarta.validation.constraints.Email;

import jakarta.validation.constraints.Max;

import jakarta.validation.constraints.Min;

import jakarta.validation.constraints.NotNull;

import jakarta.validation.constraints.NotBlank;
```

UserDto {

@NotNull

=

=

name;

@Email

@NotBlank

email;

getName() {

return name;

}

name) {

= name;

```
    }

getEmail() {

return email;

    }

email) {

= email;

    }

}
```

As you can see, the class below is a **Data Transfer Object** and has the name, that does not accept a null value, a word size less than 3 and higher than 15. The email does not accept a null value nor a word that does not match the email pattern. Let's look at the following RESTful resource:

```
public class UserResource {

@POST

    @Consumes(MediaType.APPLICATION_JSON)

    public Response save(@NotNull UserDto user){




    }

}
```

As you can see, the user parameter from the **save** method is annotated with because it does not accept the requests without the user information.

With this, to test the code, you can send a request without the data or with a wrong data, and the server should return **Bad** Look at the following example using the **curl** command:

curl -X POST 'http://localhost:8080/samplejakartabeanvalidation-1.0-SNAPSHOT/resources/users' -H 'Content-Type: application/json' -v

The following is the output:

```
> POST /samplejakartabeanvalidation-1.0-
SNAPSHOT/resources/users HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
>
< HTTP/1.1 400 Bad Request
< Connection: keep-alive
< validation-exception: true
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 47
```

Note that the data was not passed and the return was 400

Validating _persistence

The Jakarta Bean Validation has a nice integration to Jakarta Persistence and permits us to declare the constraints to the entities that are validated when these entities are persisted. In the case of tables autogenerated by the Jakarta Persistence, these tables generate the database constraints according to the constraints defined at the entity. As an example, if we define an attribute as not null, the column will be not null in the database as well.

To show the integration between the Jakarta Bean Validation and the Jakarta Persistence, we will improve the example shown earlier and add an entity called **User** and a stateless session bean called Let's look at the following code:

```
import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Id;

import jakarta.validation.constraints.Email;
```

```
import jakarta.validation.constraints.Max;

import jakarta.validation.constraints.Min;

import jakarta.validation.constraints.NotNull;

import jakarta.validation.constraints.NotBlank;

@Entity

{

@Id

@GeneratedValue

id;

@NotNull

@Min(value = 3)

@Max(value = 15)



@Column

private String name;
```

@Email

@NotBlank

@Column

private String email;

getId() {

return id;

}

public void id) {

= id;

}

}

Note that the entity is very similar to the but now it has an attribute to represent the ID. However, the main point here is the constraints used with the **@NotBlank** and

The following is the **UserService** code:

```
@Stateless

public class UserService {

@PersistenceContext

    private EntityManager entityManager;

User user){



    }
```

Note that the user parameter from the save method is annotated with

Now, we need to update the **UserResource** to use the Look at the following example:

```
import jakarta.inject.Inject;
```

```
import  jakarta.validation.constraints.NotNull;

import  jakarta.ws.rs.Consumes;

import  jakarta.ws.rs.POST;

import  jakarta.ws.rs.Path;

import  jakarta.ws.rs.core.MediaType;

import  jakarta.ws.rs.core.Response;

import  net.rhuanrocha.dto.UserDto;

import  net.rhuanrocha.entity.User;

import  net.rhuanrocha.service.UserService;

import  java.net.URI;


{


userService;
```

```
user){

User userToSave = new

        user.setName(user.getName());

        user.setName(user.getEmail());

        userService.save(userToSave);



                .build();


    }

}
```

Note that we now injected the **UseService** and used it to save the data.

To test the new codes, you can execute the following **curl -X POST 'http://localhost:8080/samplejakartabeanvalidation-1.0-SNAPSHOT/resources/users' -H 'Content-Type: applicaton/json' -d {"name":"Rhuan Rocha"}' -v**

Note that we don't inform the email. It should generate a **Bad**

Creating a custom constraint

The Jakarta Bean Validation provides some good constraints out-of-the-box, as described earlier in this chapter; however, sometimes we need a constraint that the specification does not provide. It is very common in the real world, and to some, that specification provides a way to create custom constraints. Thus, we can create our constraints and reuse them in any part of the application.

To create a custom constraint, we need the annotation to bind the constraint to the attribute or method, and the code to be executed, and apply the validation. To show you an example of custom constraint, we'll create a scenario where we want to validate whether a number received via **POST** is even or odd. To do this, we will first create the annotation that will be used to declare the constraint. Look at the following example:

```
import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;
```

```
import java.lang.annotation.Documented;




@Constraint(validatedBy = EvenValidator.class)

@Documented

public @interface Even {



default

default {};

default {};

}
```

In the preceding code, we are creating the annotation **Even** to tell the Jakarta context that the value should be even. Note that we define a default value to the message, group, and payload.

Now, we need the code to apply the validation. The following is the code to validate if the number is even or odd:

```
import jakarta.validation.ConstraintValidator;

import jakarta.validation.ConstraintValidatorContext;

{

@Override

constraintAnnotation) {

    }

@Override

number, ConstraintValidatorContext constraintValidatorContext) {

==

        }

return ==

    }

}
```

In the preceding code, we define the initialize method, with the **Even** annotation as a parameter, and the **isValid** method, which has the code to validate the value.

Now, we will create a Jakarta RESTful resource called **EvenResource** to receive a **POST** request and print the value, as shown in the following code snippet:

{

@POST

public Response print(Long number){

```
    }

}
```

To test it, you can use the following **curl** command:

```
curl -X POST 'http://localhost:8080/samplejakartabeanvalidation-1.0-SNAPSHOT/resources/even' -H 'Content-Type: application/x-www-form-urlencoded' -d 'number=3' -v
```

Note that this request should return the **Bad** as the number value is 3 and is odd.

Customizing validator messages

As we saw, the constraint has a default message, and all the specification defines that all the constraints should be a default message to show when an invalid value is identified. However, sometimes we need to define our message that is more according to the business rules. The Jakarta Bean validation provides some interesting mechanisms to define a new message, and we will see it here. To help us see the new messages, we will improve our previous example to use the to intercept the message and return it as Look at the following example:

import

{

messages;

MessageDto of messages){

MessageDto message = new MessageDto();

message.messages = messages;

return message;

```
    }

}
```

In the preceding code, we have the which is used to represent the message error that will be answered as as shown in the following example:

```
import jakarta.validation.ConstraintViolationException;

import jakarta.ws.rs.core.Response;

import jakarta.ws.rs.ext.ExceptionMapper;

import jakarta.ws.rs.ext.Provider;

import java.util.List;


import java.util.stream.Collectors;

@Provider

{


e) {
```

```
messages = e.getConstraintViolations()

                    .stream()

-> constraintViolation.getMessage())



    }

}
```

As you can see, the preceding code intercepts the **ConstraintViolationException** and responds to the message as JSON. The **ExceptionMapper** is not in the scope of this chapter, and we are using it just to make it easier to test and see the results.

As the final update, we will update the RESTful resources to produce JSON, as the message is answered as JSON. Look at the following example:

```
public class EvenResource {

@POST
```

this

public Response print(Long number){

}

}

Now, we can test it by using the following **curl** command:

curl -X POST 'http://localhost:8080/samplejakartabeanvalidation-1.0-SNAPSHOT/resources/even -H 'Content-Typeoded' -d 'number=3' -v

Note that the output should be an array with an error message, as **POST /samplejakartabeanvalidation-1.0-SNAPSHOT/resources/even HTTP/1.1**

> User-Agent: curl/7.29.0
> Host: localhost:8080
> Accept: */*
> Content-Type: application/x-www-form-urlencoded
> Content-Length: 8
>

*** upload completely sent off: 8 out of 8 bytes**
< HTTP/1.1 400 Bad Request
["the number needs to be even"]

Now, we can configure a new message to the constraint. In the first part, we'll define a simple text, and in the next part, we will use the expression language and properties. Look at the following example using a simple text:

```
public class EvenResource {

@POST

    public Response print( = "The number is not Long number){

    }

}
```

Note that, now we have the message property in the **@Even** annotation. If we run the same **curl** command as we run earlier,

the new message should be returned.

Now, we want to use the expression language to add value to the text. Look at the following example:

```
public class EvenResource {

@POST

    public Response print( = "The number ${validatedValue} is not
Long number){

    }

}
```

Now we have the message property in the **@Even** annotation using the expression language that refers to the value in the. If we run the same **curl** command as we run earlier, the new message should be returned.

Now, we want to extract it to an external property file. The Jakarta Bean Validation provides the **ValidationMessages.properties** file that is used to extract the messages from the code and put them inside a separate file. Thus, we can refer to the property using the "{" at the beginning of the property name and the "}" at the end of the property name, inside the annotation. Look at the following example:

@Even(message = "{net.rhuanrochar.even.message}")

Thus, the code of the **EvenResource** will be as follows:

```
public class EvenResource {

@POST



    public Response print( = Long number){



    }

}
```

Now, we need to create the **ValidationMessages.properties** inside the resources folder and put the property there. Look at the example of the

#My validation properties

number ${validatedValue} is not even

Note that we used the same text and we can use the expression language inside the file as well.

Conclusion

The Jakarta Bean Validation is a very useful specification from Jakarta EE. It has the integration with many other specifications from Jakarta EE like Jakarta Persistence and Jakarta RESTful, which makes it easier to use them. Providing validation can be a boring task and ruin the code. The Jakarta Bean Validation avoids these problems and provides an easy mechanism to validate the data. This specification is very commonly used inside the Java ecosystem, by the Jakarta application, and others like Spring, Quarkus, and others.

Symbols

A

B

C

D

E

F

H

I

J

M

N

O

P

Q

R

S

T

U

V

W