Java EE 8 Microservices

Learn how the various components of Java EE 8 can be used to implement the microservice architecture

Kamalmeet Singh, Mert Çalışkan
Ondrej Mihályi, and Pavel Pscheidl

Packt
www.packt.com

Java EE 8 Microservices

Learn how the various components of Java EE 8 can be used to implement the microservice architecture

.

Kamalmeet Singh
Mert Çalışkan
Ondrej Mihályi
Pavel Pscheidl

Packt

BIRMINGHAM - MUMBAI

Java EE 8 Microservices

Commissioning Editor: Aaron Lazar
Acquisition Editor: Sandeep Mishra
Content Development Editor: Digvijay Bagul
Technical Editor: Aniket Iswalkar
Copy Editor: Safis Editing
Project Coordinator: Ulhas Kambali
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Tom Scaria
Production Coordinator: Deepika Naik

First published: December 2018

Production reference: 1271218

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

ISBN 978-1-78847-514-3

`www.packtpub.com`

Contributors

About the authors

Kamalmeet Singh got his first taste of programming at the age of 15, and he immediately fell in love with it. After spending over 14 years in the IT industry, Kamal has matured into an ace developer and a technical architect. He is also the co-author of a book on design patterns and best practices in Java. The technologies he works with range from cloud computing, machine learning, augmented reality, and serverless applications to microservices and more.

Mert Çalışkan is a coder living in Ankara, Turkey. He has over 10 years experience in software development in the architectural design of enterprise Java applications. He is an open source advocate for software projects such as PrimeFaces, and has also contributed to and been the founder of various others. Currently, he also works as a consultant for Payara Application Server. He is a co-author of *PrimeFaces Cookbook*, by Packt Publishing, and a co-author of *Beginning Spring*, by Wiley Publications. He is an occasional author for Oracle Java Magazine. He is the founder of AnkaraJUG, which is the most active JUG in Turkey. In 2014, he was recognized as a Java Champion for his achievements. He is a part-time lecturer at Hacettepe University on enterprise web application architectures and web Services. He shares his knowledge at national and international conferences, including JavaOne 2016, JDays 2015, JavaOne 2013, JDC 2010, and JSFDays'08. You can reach Mert via his Twitter handle: `@mertcal`.

Ondrej Mihályi is a software developer and consultant specializing in combining standard and proven tools to solve new and challenging problems. He's been developing in Java and Java EE for 9 years. He currently works as a support engineer and a developer advocate for Payara Services Ltd. As a scrum master and experienced Java EE developer, he also helps companies build and educate their development teams. He loves working with the Java EE community and is a contributor to a number of open source projects in the Java EE ecosystem, including Payara Server and Eclipse MicroProfile. He's a co-leader of the Czech Java user group and talks at international conferences such as JavaOne, Devoxx, GeeCon, and JPrime.

Pavel Pscheidl is a man of many interests. He works as a researcher in the faculty of informatics at the University of Hradec Králové. Pavel, with his focus on statistics and agent-based simulations, currently specializes in smart systems and highly parallel simulations. In addition, he is usually to be found developing for various big companies as a consultant. Pavel enjoys the beauty and simplicity of Java EE in many projects on a daily basis and does his best to pass on his knowledge by teaching students, attending conferences, and giving talks. He is also a passionate blogger and Java EE article writer.

About the reviewer(s)

Aristides Villarreal Bravo is a Java developer, a member of the NetBeans Dream Team, and a Java user groups leader. He lives in Panama. He has organized and participated in various conferences and seminars related to Java, JavaEE, NetBeans, the NetBeans platform, free software, and mobile devices. He is the author of *jmoordb* and tutorials, and he blogs about Java, NetBeans, and web development. Aristides has participated in numerous interviews on sites about topics including NetBeans, NetBeans DZone, and JavaHispano. He is a developer of plugins for NetBeans. He is the CEO of Javscaz Software Developers. He has also worked on *Developers of jmoordb*.

I would like to thank my mother, father, and all my family and friends.

Table of Contents

Preface

Microservices is one of the hot topics in today's IT world. Though the topic is popular, it is not perfectly understood the majority of the time. Everyone seems to have a different understanding of the concept. This book tries to simplify the concept and discuss it in a manner that everyone can understand. It talks about the various basic concepts that each microservices-based application should implement.

The book starts with an introduction to microservices, and explains why it is *de rigueur*. It dwells on the existing challenges being faced by the software industry, and how microservices can help us to solve them. The book focuses on the Java-based implementation of microservices, and will help readers to explore different ways of creating them.

The book covers core concepts, such as creation, scaling, securing, monitoring, building, deploying, documenting, and testing microservice-based applications. With each of these concepts being covered, the book provides best practices that can be followed by developers.

Who this book is for

This book is for anyone who works with microservice-based architecture, or who would like to learn about it. It covers basic concepts, such as understanding microservice-based architecture and various ways of implementing a microservice in Java. Then, it moves onto advanced topics, such as how to create communication channels among microservices, managing scalability and security in an application based on microservice architecture. The book also covers topics on monitoring, deploying, documenting, and testing a microservice-based application.

What this book covers

`Chapter 1`, *From Monoliths to Microservices*, covers a brief history of software development, starting with monolith application design, through to microservice-based design.

`Chapter 2`, *Creating Your First Microservice*, introduces the tools that will be used throughout the book, followed by a hands-on exercise involving Microservice implementation.

`Chapter 3`, *Connecting Microservices Together*, discusses how microservices can interact with one another.

`Chapter 4`, *Asynchronous Communication for Microservices*, focuses on asynchronous communication between microservices, and how can it be implemented.

`Chapter 5`, *Path to Robust Microservices*, discusses how, in the world of microservices, the performance and overall availability of one service may affect the performance of other dependent services. This chapter focuses on developing applications for survival in the real world.

`Chapter 6`, *Scaling Microservices*, demonstrates how to design applications to take full advantage of the Microservice architecture when optimizing the application for unexpected loads.

`Chapter 7`, *Securing Microservices*, covers the ways of securing Microservices by enabling authenticated and authorized clients to consume the endpoints in question.

`Chapter 8`, *Monitoring Microservices*, discusses how the reader is going to learn how to gather and view data regarding application performance and health over time.

`Chapter 9`, *Building, Packaging, and Running Microservices*, examines the various methods of building, packaging, and distributing Java EE Microservices.

`Chapter 10`, *Documenting and Testing Your Microservices*, deals with various tools that are available for documenting and testing microservices.

To get the most out of this book

Readers with prior experience of Java stand to gain the most from this book. It is recommended that readers should try to explore and play around with the code examples provided in the various chapters.

Download the example code files

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

You can download the code files by following these steps:

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

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

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

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

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

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
html, body, #map {
 height: 100%;
 margin: 0;
 padding: 0
}
```

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

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
$ mkdir css
$ cd css
```

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

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

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

1
From Monoliths to Microservices

Welcome to the exciting world of Microservices! In this chapter, we will try to understand what Microservices are, how and why we are shifting from age-old practice of creating a giant monolithic application to Microservices, and the advantages and challenges of using Microservices.

In this chapter,we will try to answer some of the important questions regarding Microservices, and then in the rest of the book, we will take a deeper look at things such as the creation of Microservices, security, communication, monitoring, and documentation.

In this chapter, we will cover the following topics:

- What is Monolith design?
- The challenges associated with Monolith design
- Service-oriented architecture
- Understanding Microservices
- The advantages of Microservices
- The challenges associated with Microservices

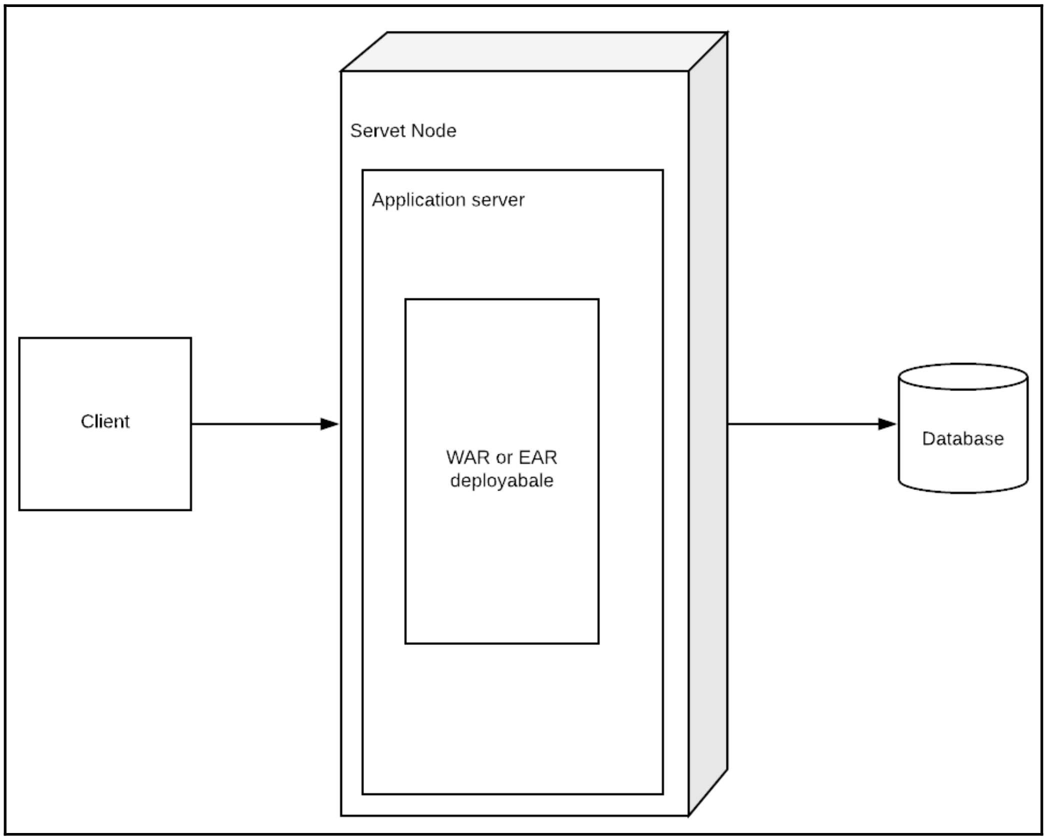
What is Monolith design?

If you have been in the industry for more than six to eight years, monolithic applications are not new to you. Even today, a lot of old applications that are in production follow monolith design architecture. Well, let's try to understand what a monolith design is and what the word monolith means to you.

If you look at these definitions and try to apply them to a software application, it will be clear what we mean when we say the application follows a monolith design. We are talking about the following characteristics:

- **Single**: In the case of an application, we are talking about a single piece of code or a deployable. An application that can and should be deployed on a single machine.
- **Indivisible**: We cannot break the application down; there is no easy way to divide the application code or deployable.
- **Slow to change**: This is more of an implication of monolith design. It is a well-known fact that changing a small piece of code is easier than a big, monolith code, especially since you don't know what implications such a change will have.

The following diagram shows the architecture of a monolithic design based on a Java application:



We can see the whole application is deployed as a single deployable, that is, a WAR or EAR file. The design looks very simple, but it does hide a lot of complexities. The single deployable, in this case, a WAR file or EAR file, might have a lot of functionality implemented.

Let's take an example: say we have to implement an application for our company where we can manage data for employees. Let's call this the **Employee Management System**, which should be able to handle core requirements such as maintaining employee data, project data, hierarchy, attendance, leave data, salary, and financial information. In addition to these core requirements, our application would also handle additional requirements, such as reporting needs for management.

In a monolith design, we code all this functionality and build it into a single deployable. If you have been in the software industry for more than 10 years, you have probably seen and worked on some applications that follow this design. The design has been good for small and simple applications, but as the complexity of the application increases, it gets challenging to manage a monolith design.

The challenges associated with Monolith design

The following challenges make the monolith design unsuitable for large applications:

- **Huge code base**: As we are developing the application as a single unit, everything is placed under a single code base. In our previous example, we have all the Employee-Management-related functionality in a single code base. So, even if we are making a small change, such as updating a tax slab for employee salaries, we have to take the whole code for our Employee Data Management project and build the whole application instead of just the tax-related part.
- **Testing**: As the application is managed as a single unit, we need to test the whole application, even if a small change is made, to make sure there are no integration or regression issues.
- **Availability**: Let's say that, while updating an employee data report, a developer introduced an error that caused the system to run out of memory, which will bring down the whole system. So a report that actually might not add too much value to the system and may be used rarely by users has the capability of bringing down the whole system.

- **Scalability**: The code is deployed as a single unit, hence we can only scale the application as a whole, making it a heavy operation. For example, if we just need to execute multiple instances of salary processing on pay day, we cannot do that in isolation; and we need to scale the whole application by providing more hardware firepower (vertical scaling) or make copies of the whole application on different machines (horizontal scaling).
- **Inflexible**: Say we need to create a special reporting feature and we know a tool or framework is available in a different language than we use. It is not possible in a monolith architecture to use different programming languages or technologies. We are stuck with the original choice, which might have been made years ago and is not relevant anymore.
- **Difficult to upgrade**: Even a small decision, such as moving from Java EE 6 to 8, would mean that the whole application, along with all the features, needs to be tested and even a small problem would hinder the upgrade, whereas if we were dealing with multiple small services, we could upgrade them independently and sequentially.
- **Slow Development and time to market**: For these reasons, it is difficult to make changes to an application with a monolith design. This approach does not fit well in today's agile world, where customers need to see changes and modifications as soon as possible, rather than waiting for a release that takes years.

We have talked about some of the challenges of monolith applications. As the application size grows and it becomes complex, it is not possible to manage a monolith application easily. Due to these challenges, the industry has explored different approaches to manage applications, with Microservices being a very popular solution.

Service-oriented architecture

Before moving to Microservices, it is important to understand what **Service-oriented architecture** (**SOA**) is. SOA is the base on which Microservices are built. As the name suggests, SOA is about services. With SOA, we try to visualize and design the application as a combination of services that can talk to each other and to the end user, and fulfill the user's requirements.

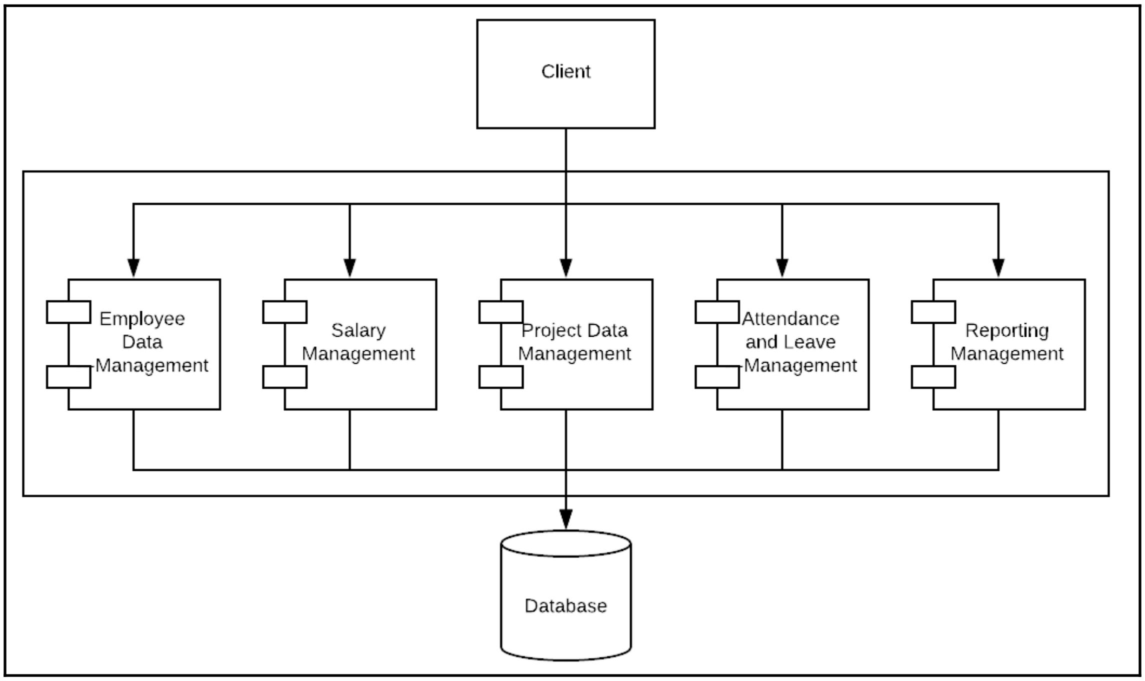
If we go back to our Employee Management System example, we can try to visualize the application as a set of different services:

- Employee Data Management
- Salary Management
- Project Data Management

- Attendance and Leave Management
- Reporting Management

Based on our needs, we can divide the application into various services.

The following diagram should help us to visualize our application:



Again, we do not need to get into complex decisions at this point, such as whether we should deploy these services as a single application or keep the data in a single database. Instead, we would like to emphasize the core idea in this section. The base of SOA is trying to think and visualize your application as a set of different services rather than a single, monolith deliverable.

Why does breaking the application into services help? A simple example is when we need to make changes to the code for salaries, the developer does not need to worry about what is happening in Employee-, project-, or attendance-related code. Also, if we were to get an error while generating a report for a manager, we would know we need to start looking for a problem in the reporting service. While testing, we will focus on only one aspect of the system. So the code becomes easy to manage and maintain.

Once we understand SOA and start to think of our application as a group of services, we can take the next step and move toward a Microservices-based architecture, which we'll discuss in the next section.

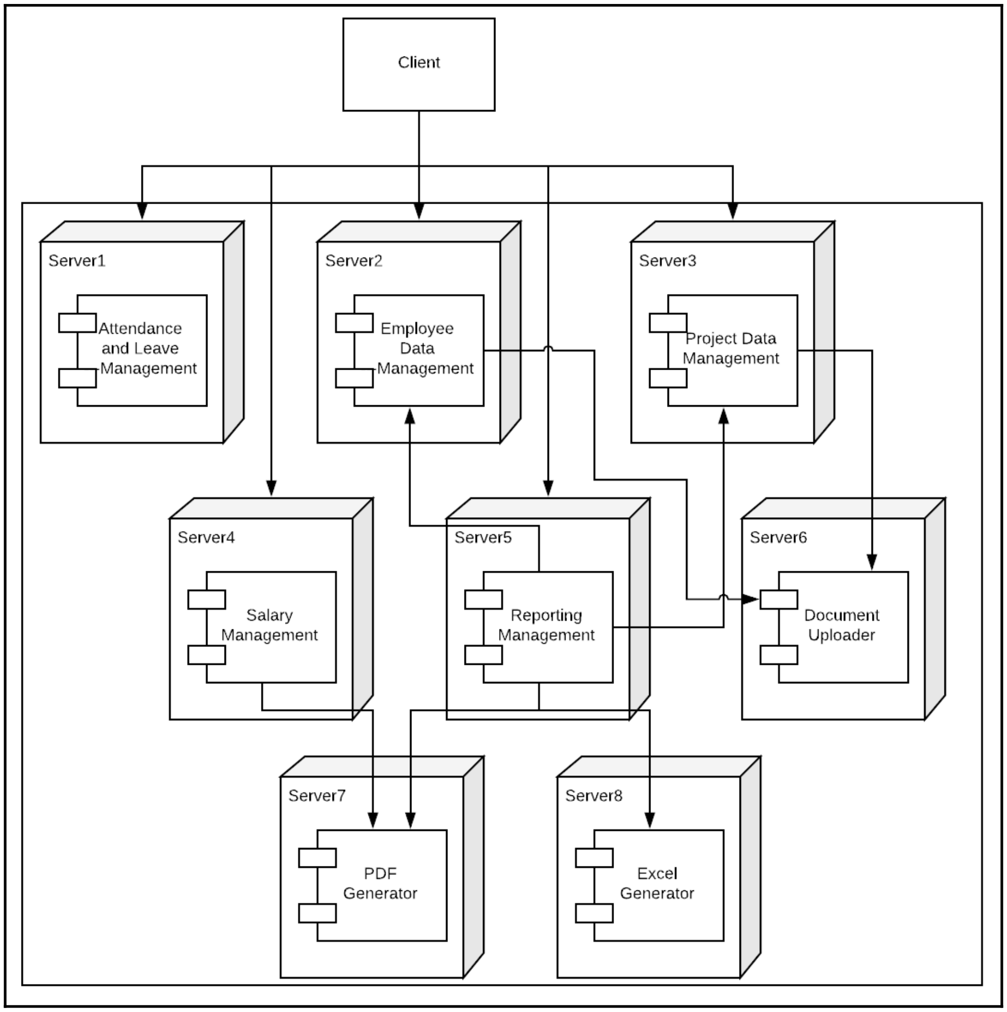
Understanding Microservices

Microservices are not a completely new concept. In the past, there have been many attempts, such as EJBs,**Remote procedure calls** (**RPC**), and implementations of services through SOAP, that aimed to reduce dependencies among various application components. Let's look at a formal definition to set the context and then we will try to understand it in detail:

> *"Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack".*
>
> `-http://microservices.io/`

In the previous section, we discussed SOA. Microservices are the logical next step, where we extend the SOA and start thinking about dividing the services at a granular level. As the name suggests, we divide our services down to a Micro level when we are talking about Microservices. The next most important aspect is to think about these Microservices as independent entities that can be developed, tested, deployed, and managed as complete sub-applications in themselves.

If we try to visualize our previous example of the Employee Management System in terms of design, the following diagram should demonstrate a Microservices-based implementation:



Let's take a look at the design and how is it different from our previous SOA approach. First, notice that we have divided our services at a granular level. For example, we are supporting Excel and PDF-based reporting, but the actual generators need not be part of the reporting service. The reporting service should only be responsible for generating data, but how the data is represented should not be part of this service. An additional advantage it gives us is that, if in future, we want to support other formats, say a Word DOC report, we don't need to touch any of the existing code; we just create a new service, say Word Document report generator, and plug it in. So we can say that a Microservices-based architecture is easy to extend and maintain.

As we have divided the application into smaller services that can be managed independently, we also talk about smaller teams and decentralized management. Each team can take independent decisions on the design and technology stack they want to use, and hence there is no dependency on other teams. Each service can be easily tested independently.

Another thing you might have noticed is that each service is **deployed independently**. Well, this might not be the exact scenario and you might deploy multiple services on the same machine, but the idea is that you should have the capacity to deploy each service independently.

Though we have not looked at the data storage part, ideally, each service will manage its own data. There should be a single point of managing one data entity. This helps in the decentralization for data, which helps with easy scalability and maintenance.

Communication between the services is another important aspect you should consider when choosing a Microservices-based architecture. We need to decide whether the communication needs to be synchronous or asynchronous, through REST, Queue-based, or some other communication medium. A well-architected system will be fault-tolerant, as a failure in no single service can bring down the system as a whole, so there is no single point of failure.

There are no fixed, industry-wide set of rules to follow for a Microservices-based architecture. This causes a lot of confusion, as well as flexibility. But to keep it simple, let's take a look at some of the common characteristics of a good Microservices-based architecture:

- Decoupled architecture
- Independent deployables
- Decentralized data
- Smaller code bases and teams
- Decentralized management
- Fault-tolerant

Next, let's take a look at some of the advantages and challenges that can be expected when using a Microservices-based architecture.

Advantages of Microservices

Now that you're comfortable with the concept of Microservices, let's take a look at the advantages they provide, which has made them so popular over the last few years. We already discussed the challenges that come with a monolithic architecture. Most of the challenges of Monolithic applications can be handled by the use of a Microservices-based approach. The following are some of the advantages of a Microservices-based architecture:

- **Easy-to-manage Code**: As we are able to modularize and divide our huge application code base into various Microservices, we are not dealing with the whole application code at any point in time.
- **Flexibility of choosing the tech stack**: As every Microservice we create is potentially a separate application in itself with a different deployable, it is easy to choose a technology stack based on need. For example, if you have many Java-based services in an application, and a new requirement comes in which you feel can be easily handled by using Python instead of Java, you can go ahead build that service in Python, and deploy it without worrying about the rest of the application.
- **Scalability**: As every Microservice can be deployed independently, it is easy to scale them without worrying about the impact on others. For example, let's say we know the reporting service is used heavily at every end of a quarter – we can scale only this Microservice by adding more instances, and the rest of the services remain unaffected.
- **Testing**: Unit testing, to be more specific, is easy with a Microservices-based approach. If you are modifying the leave-management service with some new rules, you need not worry about other services, such as Employee Project Management. In the worst case, if your leave-management service breaks down due to faulty code, you can still edit and update the Employee project-related information without even knowing that some other service is broken.
- **Faster time to market**: As we are dealing with only one part of the application at a time, it is easier to make changes and move to production. Testing and deployment efforts are minimal as we are dealing with a subset of the whole system at a time.
- **Easy to upgrade or modify**: Let's say we need to upgrade a Microservice, upgrade software or hardware, or completely rewrite the service, this is much easier in a Microservice based architecture, as we are only upgrading one part of the application.

- **Higher fault tolerance**: In a monolith architecture, one error can cause the whole system to crash. In a Microservice-based architecture, in the worst case, a single service will crash, but no other services will be impacted. We still need to make sure we are managing errors properly to take advantage of Microservice-based architecture.

Challenges with Microservices

Before concluding the Microservices section, it's important to mention that Microservices are not a silver bullet. Along with all the advantages that come with Microservices, we need to be aware of the challenges that they bring if not used properly:

- **The right level of modularization**: You need to be very careful in determining how your application can be divided into Microservices. Too few would mean you're not getting the proper advantage of Microservices, and too many services means a heavy dev-ops requirement, to make sure all the Microservices work well when deployed together. Too many Microservices can also have a performance impact due to inter-service communication needs. You need to carefully analyze the application and break it down into logical entities, based on what would make sense to be thought of as a separate module.
- **Different tech stacks to manage**: One of the advantages of a Microservices-based architecture is that you are not dependent on one technical stack or language. For example, if one of the services is coded in Java, you can easily build another one in .NET or Python. But if you are not careful, this advantage can quickly become a problem. You might end up supporting dozens of technical stacks and managing expertise for each service independently. Movement of team members between projects or among teams is not an option if required, as one team might be working on a completely different tech stack than other.
- **Heavy reliance on Dev-Ops**: If you are using too many Microservices, you need to monitor each one and make sure all the communication channels are healthy at all times.
- **Difficult fault management**: If you have dozens of services communicating with each other, and one of those goes down or is slow to respond, it becomes difficult to identify the problem area. Also, you do not want that problem in a single service to impact other services, so you will need to make sure arrangements are in place to handle error situations.
- **Managing the data**: As a rule of thumb, we try to make sure every Microservice manages its own data. But this is not always easy when data is shared among services, so we need to determine how much data each service should own and how the data should be shared among services.

When you are architecting the system, you need to make sure you understand these challenges and take care of them before committing to a solution. We will discuss some of these challenges in this book and approaches to handling them.

Summary

In this chapter, we discussed the basics of Microservices. We started by discussing Monolith architecture, and challenges that make it unfit for larger applications. After that, we discussed the SOA, which can safely be considered as the basis of Microservices. Finally, we talked about a Microservices-based architecture, as well as its characteristics, advantages, and challenges.

Throughout the rest of the book, we will look at different aspects of Microservices; we will talk about development, security, monitoring, and other aspects that you will need to be aware of when you are implementing Microservices. In the next chapter, we will discuss developing your first Microservice.

2
Creating your first Microservice

A Microservice created with Java EE or Spring Boot is easy to implement and fast to code. The amount of code a developer needs to write while implementing a Microservice is very little, as most of the boilerplate code is provided by Spring Boot or Java libraries. However, before a Microservice can be created in both technologies, a few concepts need to be understood.

In this chapter, the process of building the Microservice project from scratch is described in detail. First, the tools required for project creation and management are described and explained. Having the tools installed, the second part of this chapter is a thorough guide on project creation. With your project created, two dedicated sections follow, providing a thorough hands-on experience of coding a similar Microservice both with Java EE and Spring Boot. A simple way to run the Microservices is then described.

The aim of this chapter is to provide the reader with the basics of Microservices creation on the Java platform, be it with Spring or Java EE. Each step is described in detail to make sure that the reader understands the concepts well.

The topics to be covered in this chapter are listed as follows:

- Setting up the development environment
- Installing the Java Development Kit and Apache Maven
- A Java EE Microservice
- A Spring Boot Microservice
- Creating a project with Maven
- Building and running the Spring Boot Microservice

Setting up the Development Environment

When starting to develop both Java EE and Spring Framework applications, there are few prerequisites. As both Spring Framework and Java EE are based on Java, the installation of the artifacts given in the following steps can be used for both:

- Installing a Java development kit
- Choosing a Project management and Build tool
- Integrated development environment

In fact, the only step that is truly required is the first one – installing the **Java Development Kit** (**JDK**). JDK is the technological essence of every Java EE or Spring application. It contains many crucial parts assembling the resulting applications and, among them, the **Java Compiler** (**javac**) dominates.

However, the process of application assembly, be it a plain Java application, Java EE application, Spring Framework application, or any other framework, is represented by a complex set of steps. Remembering the sequence of steps is complex and unnecessary. A well-chosen Project management and build tool automates the process of application assembly into a single command. This leads to trouble-free and lightning-fast development. In reality, it is recommended to use a dependency management tool for every project. Among other lesser-used options, there are two important Project management tools:

- Apache Maven (`www.maven.apache.org`)
- Gradle (`www.gradle.org`)

Creating Microservices with both Java EE and the Spring Framework does not strictly depend on any build tool. For the sake of simplicity, Apache Maven is going to be used throughout this book. Apache Maven is well established in the Java ecosystem and has wide support among **Integrated Development Environments** (**IDE**).

The IDE provides assistance during the process of creating the application itself, including further integration with Project management tools, Code generation, and Auto-completion. This book respects the choice of each and every reader and is written as completely IDE-invariant. The code presented in this book relies on JDK and Apache Maven as a build and dependency management tools.

If you are new to the Java ecosystem, there are several IDEs to choose from:

- Apache NetBeans
- Eclipse IDE/Spring Tool Suite
- IntelliJ IDEA

Apache NetBeans, an IDE with origins back in 1996, tends to be the first choice for Java EE development. Eclipse IDE, first released in 2001, plays a significant role in the lives of Spring Framework developers. There is a special edition named **Spring Tool Suite** (**STS**). This edition of Eclipse provides extensive support for Spring application development.

IntelliJ IDEA is not an IDE available for free when it comes to Java EE or Spring Framework development. IntelliJ IDEA provides the best of both worlds of STS and NetBeans, and can be recommended for the development of both.

Environment installation

Both Java EE and Spring Boot are Java-based technologies. In order to build an application with Java, a correctly-configured development environment is a necessity. In this chapter, we will look at how to obtain and configure the basic tools to develop Microservices on top of Java EE or Spring, as well as tools to interact with Microservices. This chapter introduces the tools required for all chapters in this book.

Installing Java Development Kit

Java is a free ecosystem and there are various JDK available. Downloading the official and most-used JDKs from Oracle is advised. The latest JDK is available for download at `https://www.oracle.com/technetwork/java/javase/downloads/index.html`. Java SE is guaranteed to be backward compatible, thus it is recommended to always download the latest version. Java is available for all major operating systems, be it Linux, Windows, macOS, or Solaris.

On Windows, the installation of JDK is a simple process accompanied by an installation wizard. For macOS developers, there is an Apple Disk Image available to be mounted. On Linux, due to various distributions and package managers specifically, JDK is not available for each and every one of them. For distributions with the RPM package manager, there is a package prepared. For other distributions, there is a generic package made available by Oracle. The official JDK download site mentioned earlier in this chapter gives sufficient information on the downloading and installing of the JDK on a given operating system.

There is one goal in common for users of every operating system: to have the `JAVA_HOME` environment variable set up, pointing to the folder where JDK was installed.

On the Windows operating system, go to **Control Panel** | **System** | **Advanced Settings** | **Environment Variables**. Ensure there is a variable named `JAVA_HOME` with a value properly set to JDK. On Windows, JDK is most commonly installed under `C:\Program Files\Java\jdk{version}`. On Linux, the status of the `JAVA_HOME` variable can be checked by opening a Terminal and issuing the `echo $JAVA_HOME` command.

If the value returned is empty, enter the `export JAVA_HOME = /path/to/jdk` command to set the value.

Installing JDK on macOS is pretty straightforward. First, you need to download the corresponding `.dmg` installation file from Oracle. The template of the file is as follows:

```
jdk-8uversion-macosx-x64.dmg
```

After downloading the `.dmg` file, double-click on it and a finder window will appear that contains the `.pkg` installer file. Click on the installer and follow the instructions to complete the installation.

To check the `JAVA_HOME` environment variable is set up correctly on macOS, edit the `.bash_profile` file that resides on the home directory, which can be accessed via ~. It should contain an entry similar to the following:

```
export
JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_version.jdk/Contents/H
ome
```

Here, the version is the current version of JDK downloaded. Also, make sure that the `JAVA_HOME` environment variable is exported into the following `PATH` environment variable. `PATH` could contain more than one definition concatenated with semi-colons:

```
export PATH=$PATH:$JAVA_HOME/bin
```

Installing Apache Maven

Apache Maven is used throughout this book to create both Java EE and Spring Boot applications, manage applications and dependencies, and run applications. It is available for any operating system that JDK is available for. IDEs tend to contain their own distribution of Apache Maven. In this book, however, for the sake of neutrality and making the examples future-proof, Maven is used directly, as a separate installation.

The tool can be downloaded at `maven.apache.org`. Downloading the latest version available is advised. Several bundles with Maven are available for download. The binary archive without sources is sufficient. In any package downloaded, executable binaries named `mvn` are to be found in the `/bin` folder. This `mvn` binary provides a command-line interface to instruct Maven and represents the very part of Maven used throughout the book. Since the interface is intended to be used from a command line, appending the path to the directory of the `mvn` binary to the `PATH` system variable is recommended. To check Maven's availability on the system path, issue the following command:

```
> mvn --version
Apache Maven 3.5.0
Maven home: /usr/share/maven
Java version: 1.8.0_131, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-8-oracle/jre
Default locale: cs_CZ, platform encoding: UTF-8
OS name: "linux", version: "4.4.0-83-generic", arch: "amd64", family:
"unix"
```

Maven is a Java-based tool. After issuing the command, Maven should print the following output, not only stating that Maven itself is available, but also indicating that Java was found in the filesystem as the `JAVA_HOME` environment variable was properly set.

Downloading development tools

The process of development may require additional tools to interact with Microservices being built.

cURL

For transferring data with URLs by means of a command-line, cURL is one of the most-used tools. Most Linux distributions already have cURL installed. By default, this tool is present on machines running macOS. Windows users can download the tool from `http://curl.haxx.se`. In this book, cURL is used to perform basic interactions with the Microservice APIs.

Postman

Postman is a tool for interacting with and building APIs. Since REST APIs are key way to implement communication with Microservices in most of the cases, Postman is a suitable tool to test these services. Postman started as a Chrome plugin, but can be downloaded as a standalone version from `www.getpostman.com`. It is suitable for interacting with RESTful APIs, and the tools are available for free. Advanced interaction with Microservice APIs is performed with and demonstrated on Postman in this book.

Creating the project with Maven

The first use of Maven is to automate the Java project's creation. The procedure for creating a basic project is the same as for Java EE and Spring. Project configuration, and types and amounts of dependencies, are different for each of them. We will see an example usage of Maven in next section, that is Your very first Microservice.

To create a project, the `mvn archetype:generate` command can be used. This command requires the project name as input:

- groupId: This identifies the project across all other projects. Often, a reversed domain name is used.
- artifactId: This represents the name of the project.
- version: This can be omitted. The default value is 1.0-SNAPSHOT.

The following code shows a sample Maven generate command:

```
mvn archetype:generate \
-DgroupId=com.packtpub.microservices \
-DartifactId=weather-service \
-DinteractiveMode=false
```

A folder named after the `artifactId` property is created by Maven. In this case, it is the `weather-service` folder. This a folder created by Maven is referred to as `.{project-root}` throughout the book. In `.{project-root}`, a simple file structure is to be found, which is shown here:

```
.{project-root}
├── pom.xml
└── src
```

The `pom.xml` file describes project properties and required libraries. Also, the packaging of the Microservice is defined in this file. In the `src/` folder, the source code of the Microservice will reside.

For the sake of simplicity, there are many nuances of Maven that are not explained in this book. The ways of generating a project with Maven are countless. Visit `maven.apache.org` to obtain a detailed introduction to Maven. In the Introduction guide, the basic concepts are explained, including the reasoning behind archetypes, snapshot versions, and naming conventions.

Your very first Microservice

In the domain of a smart city, there are many sensors measuring local climate conditions, such as temperature or humidity. Let's assume that the temperature data is used by some other systems in the city. How do these other systems obtain information about the current temperature? A dedicated service that does one thing well is a Weather Microservice. Writing the code over and over again with each new system results in low code reuse, high additional costs, and difficult maintainability. Thus, the entire logic of dealing with weather data is extracted into one Microservice that other systems may call at any time. In order for the other systems to be able to communicate with our Weather Microservice, a means of communication must be set up. Therefore, the Weather Microservice has a strictly defined application interface. In the age of the internet and Web 2.0, where HTTP as a protocol is well established and accepted, choosing a RESTful interface ensures barrier-free access to the Microservice.

The goal is therefore to create a simple Weather Microservice that provides data about the temperature in the city. The service is going to communicate with the outer world by means of a RESTful interface over an HTTP protocol.

The process of creation of this Microservice will be demonstrated with both Java EE and Spring Boot. The source code of the Microservices created in this chapter are available on GitHub. The repository contains a finished project with thoroughly commented code, ready to be run.

A Java EE Microservice

Java EE as a technology is an excellent choice for Microservices because it provides some off the shelf functionalities for implementing core requirements of communication, security, scalability, and so on. Each and every functionality from the Java EE specification can be plugged into the Microservice when needed, and dropped when it is no longer required. As there are many implementations of one specification available, support and security updates, as well as vendor neutrality for any Microservice, is ensured. There are many ways of building a Microservice with Java EE. In this introductory chapter, a minimalistic approach with Payara Micro, taking care of application runtime, is presented.

We are moving ahead with the assumption that the reader is aware of the basic usage of Maven. Though it is not a must to understand the concepts, it is recommended to go through the basics of Maven if you are new to it: `https://maven.apache.org/`.

Coding the Microservice

With the project layout created by Maven, the Weather Microservice itself can be implemented. There are two tasks to accomplish:

- Implement a resource that provides temperature information to the outside world
- Configure the application's API context path

Each of these steps requires a single class, one for the temperature resource, named `TemperatureResource`, and one for the application's context path configuration, named `WeatherApplication`.

The temperature resource, which serves information about temperature, is represented by a Java class, named `TemperatureResource`, placed into the `com.packtpub.microservices.weather.temperature` package. In Maven's project structure, the package belongs to `{project-root}/src/main/java folder`. This is the folder where IDEs will create the package as well, as follows:

```
package com.packtpub.microservices.weather.temperature;

import javax.ws.rs.core.Response;

public class TemperatureResource {
   public Response getAverageTemperature() {
       //Not yet implemented in this step
   }
}
```

There is a zero-argument method, named `getAverageTemperature()`, defined. The return type of this method is `javax.ws.rs.core.Response`. The object comes from the Java API for RESTful Web Services, or JAX-RS for short. The `Response` object returned from the method contains mostly HTTP-protocol related information about response headers, body content, redirects, and much more.

`Response` is not instantiated explicitly. Instead, the process of `Response` creation follows a builder pattern, allowing the developer to obtain an instance from one of the static methods defined on the `Response` class. The static methods return a `ResponseBuilder` instance. Once the process of building the response with the `ResponseBuilder` instance is over and the response contains everything the programmer wants, the final `Response` object is constructed by calling the `build()` method on the `ResponseBuilder` chain. The process of building a simple `Response` is demonstrated in the following code sample:

```
package com.packtpub.microservices.weather.temperature;

import javax.ws.rs.core.Response;

public class TemperatureResource {
   public Response getAverageTemperature() {
        return Response.ok("35 Degrees")
                  .build();
   }
}
```

The `TemperatureResource` of the Weather Microservice is implemented, but there are no instructions under which URI path the resource is available, or in simple words, it is not clear how to call this service. The resource's paths are declared as any additional metadata is, by means of a specialized annotation, `@Path`, from the `javax.ws.rs` package. This annotation accepts one string value, defining the path of the underlying resource. By applying the `@Path("/temperature")` annotation to the `TemperatureResource` class, JAX-RS is instructed to make the underlying set of endpoints available under the `{http|https}://{host}:{port}/temperature` URI.

The code with the resource path defined by means of a `@Path` annotation, is demonstrated as follows:

```
package com.packtpub.microservices.weather.temperature;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;

@Path("/temperature")
public class TemperatureResource {
   public Response getAverageTemperature() {
       return Response.ok("35 Degrees")
               .build();
   }
}
```

Defining the unique URI of the `TemperatureResource` class is not enough for the `getAverageTemperature()` method to be recognized by JAX-RS as a RESTful resource. First of all, the very existence of a Java method on a JAX-RS resource class does not automatically imply that JAX-RS considers such a method to be a specific resource. Java EE developers are free to create an arbitrary amount of methods inside a JAX-RS resource class without introducing any side effects. HTTP protocol declares several methods for client-server communication, as declared in RFC 2616. Each method has its own pre-defined purpose. For example, for retrieving information identified by the request URI, the `GET` method is intended to be used. Making `getAverageTemperature()` a JAX-RS resource available via an HTTP GET call means we put the `@GET` annotation from `javax.ws.rs package` on top of the method.

The final look of `TemperatureResource` is demonstrated in the following code block:

```
package com.packtpub.microservices.weather.temperature;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;

/**
* RESTful resource providing information about city's temperature
*/
@Path("/temperature")
public class TemperatureResource {

    /**
     * Provides average temperature from all the city's sensors. The
temperature
     * is artificial.
     *
     * @return {@link Response} with constant temperature
     */
    @GET
    public Response getAverageTemperature() {
        return Response.ok("35 Degrees")
                .build();
    }
}
```

The combination of annotation metadata tells JAX-RS that the `getAverageTemperature()` method is available via an HTTP GET call under the path defined by the `@Path("/temperature")` annotation. Other HTTP methods are also available in the same package.

JAX-RS requires the developer to specify the URI path of resource classes exactly, as JAX-RS does not try to guess or derive the URI from the resource class name or from the name of any of its methods. Developers are free to name classes and methods deliberately. This results in zero interference between Java code and the RESTful interface of the Microservice.

Configuring the Microservice

After the temperature resource has been finished, it is necessary to define the context path of the final URL where the temperature resource will be available. Start with creating the JAX-RS configuration class, named `WeatherApplication`, in the `com.packtpub.microservices.weather` package. In order for `WeatherApplication` to be recognized as a JAX-RS configuration class, it must extend the `javax.ws.rs.core.Application` class. The configuration of the application's path is achieved with the `javax.ws.rs.ApplicationPath` annotation. The annotation takes a String that represents the context path root of the application's resources. Annotating the class with `@ApplicationPath("/weather")` tells the JAX-RS implementation to expose all resources under the `/weather` base context path. The slash in the path is optional; its absence is automatically resolved:

```
package com.packtpub.microservices.weather;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

/**
* Configuration of Weather Service's REST API's context path.
*/
@ApplicationPath("/weather")
public class WeatherMicroService extends Application {
}
```

Code summary

A Weather Microservice with a basic RESTful endpoint providing information about a city's average temperature has been created. There were two classes created, each with a unique role in the system:

- `WeatherMicroService`
- `TemperatureResource`

The `WeatherMicroService` class holds a minimalistic configuration of the Weather Microservice. It is required to explicitly state the basic context path of the Microservice URI.

The `TemperatureResource` class represents a specific resource within the Microservice itself. It contains one endpoint returning the mentioned average temperature in the city, represented by the `getAverageTemperature()` method. We introduced two additional pieces of metadata written in the form of an annotation:

- The `javax.ws.rs.Path` annotation, which defines the path of the temperature resource. The path is relative to the application path defined in the `WeatherMicroService` class.
- The `javax.ws.rs.GET` annotation, which tells us to user the `GET` HTTP method to make a call to `getAverageTemperature()` service.

With all these steps taken, the Microservice is entirely implemented and ready to be run.

The final outline of the project with packages and classes in `{project-root}/src/main/java` is demonstrated by the following directory tree:

```
com
    └── packtpub
        └── microservices
            └── weather
                ├── temperature
                │   └── TemperatureResource.java
                └── WeatherApplication.java
```

Running the Microservice

Java EE enables the developer to choose from many ways of running the Microservice. As the architecture of Java EE uses layers, where standards and their implementations act as one layer and the Microservice itself is built on top of the standards, the developer has many options. Among other things, it is now possible to do the following:

- Deploy to a classical application server.
- Create an UberJar that contains all the necessary functionality within one easily-runnable Java Archive.
- Create a hollow jar (which is the same as UberJar but does not contain the application code), which provides all the dependencies for the Microservice to be deployed on top of it.
- Run Microservices directly from the command line with Payara Micro.

An easy way of running Java EE Microservices or just about any Java EE applications is using Payara Micro and the command line. To download Payara Micro, visit `payara.fish/downloads`. Backward compatibility is part of the Java EE contract, thus downloading the latest version is an obvious step.

Payara Micro is a Java application that is able to run other Java EE applications. It provides a **Command-Line Interface** (**CLI**). Interacting with Payara Micro means running it as an ordinary Java program; nothing more is required. The readiness of the environment can be tested by asking for a Payara version:

```
> java -jar /path/to/payara-micro-{version}.jar --version
Payara Micro 4.1.2.173 Build Number 235
```

A list of Payara Micro's services can be displayed by means of the `--help` flag:

```
> java -jar /path/to/payara-micro-4.1.2.173.jar --help
```

Among the many options, the `--deploy` option followed by a path to the Java EE Microservice is designed to simply run any Java EE Microservices. Payara Micro invoked with the `--deploy` option is used to deploy the Weather Microservice. It even has the ability to deploy multiple Microservices at once, if required.

Building and running the Weather Microservice

Before running the Microservice, it has to be compiled and packaged. Maven, the project management tool, automates this process. The whole project is represented by Maven's `.pom` file, named `pom.xml`. This file is present in the `project's` root folder. Maven, when executed, searches for `pom.xml` in the very folder in which it has been executed. In the example application, there is only one folder and one file to be found. First and most important is the `src/` folder, with the application's code and resources. The name of this folder, `src`, is standardized by Maven unless updated to some custom folder; the sources of the application are expected to be there:

```
.{project-root}
├── pom.xml
└── src
```

In order to build the project, change the working directory to `.{project-root}`. After changing the working directory, the next step is to instruct Maven to compile, verify, and package the whole application in one instruction. After the packaging is completed, Maven reports a **BUILD SUCCESS**:

```
> mvn package
[INFO] Scanning for projects...
......several output omitted......
[INFO] ---------------------------------------------------------------------
----
[INFO] BUILD SUCCESS
[INFO] ---------------------------------------------------------------------
----
```

After `mvn package` is executed, Maven downloads all the necessary dependencies, compiles the Java code, and packages the result into a **Java Web Archive** (**WAR**). The resulting WAR is placed by Maven into a unified directory, named `target/`, in the `.{project-root}` folder:

```
.{project-root}
├── pom.xml
├── src
└── target
```

The name of the WAR placed inside the `target/` directory consists of several parts: `{artifactId}.{version}.{packaging}`. These variables are placed in the `pom.xml` file itself during the project creation phase. Maven uses them to construct the name of application artifacts during the build process.

Since Java EE is used, there is only one dependency, Java EE API. This dependency is scoped as provided, telling Maven not to include it in the resulting WAR. Therefore, no additional code is bundled in the resulting WAR, only the Microservice itself, resulting in a very small deployment unit of several kilobytes.

With the Weather Microservice built in the `/target` folder, the Microservice is now ready to be run in Payara Micro. To run the Microservice, do the following:

1. Start Payara Micro.
2. Deploy the Microservice.

Both steps can be executed with a single command:

```
java -jar /path/to/payara-micro-{version}.jar --deploy /path/to/weather-
service-1.0.war
```

By issuing that command, Payara Micro is invoked as a standard Java application and instructed to deploy a single WAR. The process of startup and deployment is automatic and does not require user interaction. The first phase is the start of Payara Micro itself. When Payara Micro is successfully started, the second phase, where deployment units are handled, is initiated. The set of deployment units is defined by the `--deploy {deployment-unit}.{war|jar|ear}` flag.

If there is a problem with running Payara Micro, the cause is usually other applications bound to ports used by Payara Micro. Typically, Payara Micro listens on port 8080, which is commonly used by many other programs.

Invoking the Microservice

For each successfully deployed application, Payara Micro prints information about the URI the application can be accessed under, as well as a list of REST endpoints exposed by the application and their URLs.

After invoking `java -jar payara-micro-4.1.2.172.jar --deploy {deployment-name}.{jar|war|ear}` from the command line, the output at the very end of the Payara Micro startup process contains the following information:

```
Payara Micro URLs
http://localhost:8080/weather-service-1.0

'weather-service-1.0' REST Endpoints
GET /weather-service-1.0/weather/temperature
```

With Payara Micro and the Weather Microservice up and running, the Microservice itself can be invoked. From the console output provided by Payara Micro, the final path, or more precisely the URL of the average temperature endpoint, can be assembled.

Payara Micro, as the majority of other Microservice solutions, binds to the localhost interface on port `8080` by default. The so-called context part of the URL is composed of several parts. The URL parts for the Temperature resource are composed as follows:

- `/weather-service-1.0` : This represents the name of the Java Web Archive deployed to Payara Micro. This part can be removed by renaming the deployment unit to `ROOT.{jar|war|ear}`.
- `/weather` : This represents the base context path for all Weather Microservice RESTful endpoints, as defined in the `WeatherMicroservice` class.
- `/temperature` : This represents the path to the final temperature resource, as defined by the `TemperatureResource` class in the code.

The final URL for the average temperature resource is derived from the preceding information to create the following URL:

```
http://localhost:8080/weather-service-1.0/weather/temperature
```

As expected, the endpoint is reachable by means of an HTTP protocol. To invoke the Microservice, a tool able to make requests with the HTTP protocol is required. A widespread utility, named cURL (to download, check out `https://curl.haxx.se/download.html`), is capable, in its own words, of transferring data to and from a server while using many protocols, and HTTP is one of them. Invoking the endpoint with cURL means passing the URL of the average temperature endpoint to the utility:

```
> curl http://localhost:8080/weather-service-1.0/weather/temperature
{"temperature":60.0,"temperatureScale":"CELSIUS"}
```

The Weather Microservice immediately responds with the temperature in JSON format.

cURL automatically recognizes the protocol from the URL passed as a parameter. Also, when no HTTP method is specified, cURL uses the HTTP `GET` method by default. As the endpoint is available via HTTP `GET`, this is desired behavior.

A Spring Boot Microservice

Spring Boot represents an evolutionary phase of the Spring Framework. Spring Boot does not introduce vastly new functionality, its focus is to make the process of creation and running of Spring-Framework-based applications easier. It is defined by a curated set of techniques and tools working together to provide a better experience throughout the application's life cycle.

Spring Boot is not restricted to Microservices. All the functionality of the Spring Framework is still available.

Creating the project with Maven

Spring Boot is a Java-based technology, therefore creating a Java project is required. Maven is going to be used as a project management tool. The process of creating a basic Java project with Maven is explained at the beginning of this chapter, under the *Setting up the development environment* section, since the process is the same for both Java EE and Spring applications. As a result, a project with an almost empty `pom.xml` file should be created:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.packpub.microservices</groupId>
    <artifactId>weather-service-spring</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>02-Smart City Weather Microservice with Spring Boot</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

Having created a basic project with Maven, Spring Framework support can be integrated by making several changes to `pom.xml`. As a bare minimum, it is required to do the following:

- Add a Spring Boot Starter parent, that is, a parent tag with group and version details
- Add Spring Boot dependencies
- Add the Spring Boot Maven plugin

Spring Boot dependencies provide a curated set of Spring Framework parts with the functionality required. The Spring Boot Maven plugin is required to package the Spring application into a **Java Archive** (**JAR**) and make it runnable. In order for the Spring Boot Maven plugin to work, additional Maven configurations and dependencies are required. However, developers are not forced to configure Maven builds to provide a suitable environment for Spring Boot to work. All the required functionality and settings are hidden in a single Maven parent, named the Spring Boot Starter parent.

Adding the Spring Boot Starter parent is the first logical step, since the rest of the steps are dependent on its presence:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.packpub.microservices</groupId>
    <artifactId>weather-service-spring</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>02-Smart City Weather Microservice with Spring Boot</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.6.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

Spring Boot applications are subject to a unique packaging system, where all the dependencies, including the optional Java EE parts that Spring is dependent on, are included and compressed into a single Java Archive. Such a JAR is then easily runnable. Maven on its own does not contain the functionality to create such a package. To create a Spring-Boot-runnable JAR, a Maven plugin is required. It is therefore necessary, after the Spring Boot Starter parent is in place, to input the Spring Boot Maven plugin, which provides the functionalities required. This plugin is added to the `<plugins>` section of the `<build>` settings section:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.packpub.microservices</groupId>
    <artifactId>weather-service-spring</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>02-Smart City Weather Microservice with Spring Boot</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.6.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>1.5.6.RELEASE</version>
            </plugin>
        </plugins>
    </build>

</project>
```

In the current state, the Spring-Boot-runnable JAR can be created, but there are no Spring Framework libraries to provide any functionality whatsoever. With Spring Boot, adding Spring Framework functionality is much easier than ever before. Spring itself has many modules with lots of useful functionality, but resolving mutual dependencies could be quite difficult at times. Spring Boot introduces a new system of dependencies, representing a more curated set of Spring Framework functionality. The goal is to create a very basic Microservice that communicates with other participants by means of a RESTful interface. With Spring Boot, this goal requires only one dependency:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.packpub.microservices</groupId>
    <artifactId>weather-service-spring</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>02-Smart City Weather Microservice with Spring Boot</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.6.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>1.5.6.RELEASE</version>
            </plugin>
        </plugins>
    </build>

</project>
```

The created `.pom` file describes a basic Spring Boot application completely. Nothing more is required.

It is also possible to use a dedicated web tool named Spring Initializr, available from `http://start.spring.io`. Spring Initializr is an interactive tool to generate Maven- or Gradle-based Spring Boot projects. For more complex projects with non-trivial functionality, project outline creation is greatly simplified. It also provides an overview of available functionality, which can be added to the project by simply clicking on a checkbox. If you would like a similar experience for your JavaEE projects, WildFly Swarm provides the same capabilities. With WildFly Swarm, you don't need to specify an exhaustive set of required dependencies, as there is automatic detection available. For more information, visit `http://wildfly-swarm.io/generator/`.

Coding a Spring Boot Microservice

To create a Weather Microservice with Spring Boot providing the actual temperature by means of a RESTFul endpoint, the following steps are necessary:

1. Create an application entry point to explicitly start the Spring Boot application.
2. Perform a basic configuration of the application.
3. Create a RESTful resource-controller class.

In the example, all classes belong to a `com.packtpub.microservices.weatherservice` package. In Maven's project structure, the package belongs to `{project-root}/src/main/java folder`. When a Java IDE is used, packages are automatically put in this folder.

Spring Boot applications require an explicit entry point in the form of the `public static void main()` method, as any Java application does. In this `main()` method, the Spring Boot application is started and configured. The beginning of a Spring Boot project is very similar to creating a Java SE application:

```
package com.packpub.microservices.weatherservice;

public class WeatherServiceApplication {
    public static void main(String[] args) {
        //Start & configure Spring Boot application
    }
}
```

The next step is to turn a common Java SE application into a Spring-Framework-enabled Java application. Spring Boot is a very convenient tool for doing that.
The `SpringApplication` class from the `org.springframework.boot` package instantiates the Spring Framework. Its public static method, `run(...)`, takes care of everything required. The `run()` method is overloaded multiple times. A simple variant covering most use cases is invoked in the `SpringApplication.run(Object[] sources, String... args)` version. The very first argument is the Class object of the Spring Boot application configuration. This points to a class holding the Spring Boot application configuration and there can be multiple such classes in a project if required. For the example project, the configuration class is the `WeatherServiceApplication` class, which by no coincidence also hosts the `main()` method. It is a common practice to unite the Spring Boot configuration class with a class hosting the `main()` method, even though it is not necessary.

As already mentioned, a Spring Framework application needs to be configured. With Spring Boot, the whole framework has evolved into a convention-over-configuration approach. This means a reasonable default configuration, with changes made only to the parts as necessary. Most of the times, there are no changes at all. The default configuration comes with a `SpringBootApplication` annotation from the `org.springframework.boot.autoconfigure` package. By applying it to the `WeatherServiceApplication` configuration class, the Spring Boot Microservice startup procedure is defined and complete:

```
    package com.packpub.microservices.weather;

    import org.springframework.boot.SpringApplication;
    import org.springframework.boot.autoconfigure.SpringBootApplication;

    @SpringBootApplication
    public class WeatherServiceApplication {

        public static void main(String[] args) {
            SpringApplication.run(WeatherServiceApplication.class);
        }
    }
```

From the example, it can be observed that the array of String arguments is a purely optional argument. Only the Class object of the `WeatherServiceApplication` configuration class is passed as an argument to the `SpringApplication.run()` method.
The `@SpringBootApplication` configuration annotation represents a union of classical Spring Framework annotations:

- `@SpringBootConfiguration`
- `@EnableAutoConfiguration`
- `@ComponentScan`

`@SpringBootConfiguration` helps Spring Boot to identify the annotated class as a configuration class. The remaining two annotations, `@EnableAutoConfiguration` and `@ComponentScan`, affect the behavior of Spring Dependency Injection and the bean-configuration mechanisms present inside a Spring application. This makes Spring Framework discover all possible beans on the classpath and attempt to resolve the dependency-injection tree automatically, without manually adding each and every bean.

With Spring Boot Microservice configured, the Weather Microservice functionality can be straightforwardly implemented. The Weather Microservice is going to provide one simple endpoint that returns an average temperature in the whole city. The service contract, that is, the method returning the temperature, promises to provide the temperature value. The service supports two scales, namely `CELSIUS` and `FAHRENHEIT`. This is the whole domain model of the Microservice. The package chosen for the entire domain model in the following examples is `com.packpub.microservices.weather.domain`, the same package as the Spring Boot main and configuration class.

There is one `enumeration` to be created, representing the temperature scales available. For simplicity, the `CELSIUS` and `FAHRENHEIT` scales are used. The enumeration is named `TemperatureScale`:

```
package com.packtpub.microservices.weather.domain;
public enum TemperatureScale {
 CELSIUS, FAHRENHEIT
}
```

Afterward, an object representation of the `temperature` itself, named the `Tempterature` class, is created:

```
package com.packpub.microservices.weather.domain;

public class Temperature {

    private Double temperature;
```

```
        private TemperatureScale temperatureScale;

        public Double getTemperature() {
            return temperature;
        }

        public void setTemperature(Double temperature) {
            this.temperature = temperature;
        }

        public TemperatureScale getTemperatureScale() {
            return temperatureScale;
        }

        public void setTemperatureScale(TemperatureScale temperatureScale) {
            this.temperatureScale = temperatureScale;
        }
    }
```

With the Object representation of the Microservice's domain model sealed, the structure of the average temperature resource is known and all the necessities for a successful average-temperature resource implementation are completed. The spring Framework leverages the functionality of the Spring REST MVC to enable developers to create RESTful services. An average temperature resource requires a new class placed in a `com.packpub.microservices.weather` package, named `TemperatureResource`. The `TemperatureResource` class hosts one method, named `getAverageTemperature()`, with a return type of `ResponseEntity<T>` from the `org.springframework.http` package:

```
    package com.packpub.microservices.weather.domain;

    import com.packtpub.microservices.domain.weather.Temperature;
    import com.packtpub.microservices.domain.weather.TemperatureScale;
    import org.springframework.http.ResponseEntity;

    public class TemperatureResource {

        public ResponseEntity<Temperature> getAverageTemperature() {
            // Not yet implemented in this step
        }
    }
```

The `ResponseEntity` class has one generic parameter, `T`, defined. This generic parameter represents the type of the Java object serialized in the HTTP response body. For the average-temperature resource, the type is `Temperature`. In other words, if there are no errors during the `getAverageTemperature()` method invocation, an instance of `ResponseEntity` containing the `Temperature` class object is set. The `HTTP 200 OK` status code is set for `ResponseEntity`. Fulfilling this contract instantiates the `Temperature` object, sets its properties, and returns it wrapped by a `ResponseEntity` object:

```
package com.packpub.microservices.weather;

import com.packtpub.microservices.domain.weather.Temperature;
import com.packtpub.microservices.domain.weather.TemperatureScale;
import org.springframework.http.ResponseEntity;

public class TemperatureResource {

    public ResponseEntity<Temperature> getAverageTemperature() {
        Temperature temperature = new Temperature();
        temperature.setTemperature(35D);
        temperature.setTemperatureScale(TemperatureScale.CELSIUS);

        return ResponseEntity.ok(temperature);
    }
}
```

The instantiation of the `Temperature` object, and the setting of its properties, are parts of a standard Java code. The created `Temperature` instance is wrapped by `ResponseEntity` with the `ResponseEntity.ok(temperature)` method. This entry tells Spring to return an HTTP response with HTTP status 200 OK and serialize an instance of Temperature in the method body. In fact, it is a convenient shortcut for `ResponseEntity.status(HttpStatus.OK).body(temperature)`. `ResponseEntity` follows a very common builder pattern, making it easy and quick to configure the response.

The Microservice functionality is defined; however, there is no way for Spring Framework to recognize, under which URL the temperature resource is available. First, Spring has to be told that the `TemperatureResource` class is a RESTful controller by using the `@RestController` annotation. A rest controller is an internal naming related only to the Spring Framework, marking a class as a web controller and marking a method's return value as serialized in the HTTP response body.

The origin of this annotation can be found in Spring MVC, a mature Spring component supporting the famous Model-View-Controller pattern. Spring REST MVC builds on top of this historical functionality. In reality, `@RestController` combines two older annotations: `@Controller` and `@ResponseBody`.

The basic context path of all endpoints regarding the temperature resource should be `"/temperature"`. To achieve this, a `@RequestMapping` annotation from the `org.springframework.web.bind.annotation` package may be used. The average temperature endpoint is available via the HTTP `GET` method, as defined in RFC 2616. To define the state method, the `@RequestMapping` attribute is used once again, this time the method attribute is filled with the GET value from the `RequestMethod` enumeration, present in the same package as the `@RequestMapping` annotation itself:

```
package com.packpub.microservices.weather;

import com.packtpub.microservices.domain.weather.Temperature;
import com.packtpub.microservices.domain.weather.TemperatureScale;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(path = "/temperature")
public class TemperatureResource {

    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<Temperature> getAverageTemperature() {
        Temperature temperature = new Temperature();
        temperature.setTemperature(35D);
        temperature.setTemperatureScale(TemperatureScale.CELSIUS);

        return ResponseEntity.ok(temperature);
    }
}
```

It should be noted that method namings are not considered in any way when Spring REST MVC assembles the final URL to the endpoint. The `getAverageTemperature()` method can be renamed without any impact on the RESTful interface whatsoever.
With `TemperatureClass` being implemented and the required Spring REST MVC metadata in a form of annotations being added, the final step has been reached and the Weather Microservice implemented with Spring Boot is ready to be run.

The final outline of the project using the example package naming is demonstrated in the following filesystem tree:

```
.
├── main
│    ├── java
│    │      └── com
│    │        └── packpub
│    │          └── microservices
│    │            └── weather
│    │            ├── domain
│    │            │    ├── Temperature.java
│    │            │    └── TemperatureScale.java
│    │            ├── TemperatureResource.java
│    │            └── WeatherServiceApplication.java
```

Building the Spring Boot Weather Microservice

Spring Boot applications are primarily distributed as a standard Java application packed inside a JAR. The JAR of any Spring Boot application contains all the dependencies required for Spring Framework applications to be run. To build a Spring Boot application, Maven can be utilized. The Spring Boot Maven plugin, which has been present in Maven's `pom.xml` file since the development phase, automatically hooks onto the build process and creates a directly executable Spring Boot JAR.

The whole process is triggered by instructing Maven to package the application. Executing the `maven package` command should be either done on the `{project-root}` folder, or by pointing Maven to `{project-root}` using the `mvn package -f path/to/pom.xml` parameter. `{project-root}` stands for the folder where the `pom.xml` of the Spring Boot Weather Microservice resides:

```
> mvn package
...several output omitted...
[INFO] ------------------------------------------------------------------------
----
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
----
```

After Maven is instructed to package the application, a new folder appears in `{project-root}`. In this folder, the Spring Boot Microservice build can be found:

```
.{project-root}
├── pom.xml
├── src
├── target
```

Inside the `target/` folder, several sub-folders and files emerge. Most of the folders are a standard output of the Java application-compilation and -assembly process, such as generated sources or Javadoc documentation. The structure of the `target/` folder is demonstrated here:

```
. {project-root/target}
├── classes
├── generated-sources
├── maven-archiver
├── maven-status
├── weather-service-spring-1.0-SNAPSHOT.jar
└── weather-service-spring-1.0-SNAPSHOT.jar.original
```

Two files are of particular interest regarding Spring Boot: there are two JAR to be found. Those JARs almost have the same name, but one of them is suffixed with the `.original` string:

- `weather-service-spring-1.0-SNAPSHOT.jar`
- `weather-service-spring-1.0-SNAPSHOT.jar.original`

The suffixed original JAR is built first. This JAR is the output of the standard Maven build pipeline when the `MVN` package command is invoked. This archive is only a few kilobytes in size, containing nothing but the business logic of the application. There is no Spring Framework library included. This archive is taken by the Spring Boot Maven plugin and transformed into a new JAR, called **Fat JAR**. The old one is then kept aside at the same location with the afore-mentioned `.original` suffix.

A Fat JAR is a common name for a Java application packed in a single JAR with all the dependencies required. The process of transforming a simple Java Archive into a Spring Boot Fat JAR includes importing the required libraries into the newly-created JAR. Everything from Spring Framework libraries to the Java EE specification implementations that Spring Framework depends on is included. With Spring Boot, the Spring Framework core principles remain. Years ago, Spring Framework applications were dependent on several Java EE specifications, most noticeably on the servlet specification. As a result, Spring Framework was commonly deployed to an Application Server or a Servlet Container. This dependency on Java EE remains. However, the days of direct interaction with a servlet container, which is just enough for Spring Microservice to run, are over. There are three servlet containers to choose from with Spring Boot:

- Apache Tomcat (`tomcat.apache.org`)
- Eclipse Jetty (`eclipse.org/jetty`)
- Undertow (`undertow.io`)

Each of these servlet containers has specific strengths. By default, Apache Tomcat is included in the Spring Boot Fat JAR. However, Spring Boot takes care of the instantiation and configuration. You no longer need to start an application server or a servlet container and then deploy a Spring application. The startup configuration and deployment steps are fully automated.

Because Spring Boot represents a way of application packaging and distribution, a common Spring Boot application can always be converted into a traditional Spring Framework application and deployed as a Web Archive into a Servlet Container or Application server. This can be achieved by editing Maven's POM, modifying the dependencies and way of packaging.

Running the Spring Boot Weather Microservice

Once the Spring Boot Microservice is developed and assembled, it is ready to run. There are several ways of running a Spring Boot Microservice:

- Executing a Spring Boot Java Archive in a JVM
- Using Maven to run the Spring Boot application
- Using the Spring Boot CLI

Since the Spring Boot Maven Plugin produces a JAR with all the dependencies found in the `{project-root}/target/` folder, an obvious choice is to run it with `java -jar weather-service-spring-1.0-SNAPSHOT.jar`. As a necessity, Java must be available on the PATH, otherwise, a path to the `java` binary folder must be provided explicitly:

```
> java -jar weather-service-spring-1.0-SNAPSHOT.jar
...several output omitted...
Tomcat started on port(s): 8080 (http)
Started WeatherServiceApplication in 2.305 seconds (JVM running for 2.601)
```

As a default behavior, Spring Boot starts a Tomcat container included in the Java Archive, registers Spring-Framework-specific servlets, and constructs application context. The default interface Apache Tomcat listens to is 127.0.0.1 (localhost) using port 8080. If the Spring Boot application is successfully deployed into the bundled servlet container, an information message about the successful start of an application appears.

The initial ASCII art logo of Spring Boot can be disabled in many ways. One simple way is to construct the Spring Boot application using the `SpringApplicationBuilder` class from the `org.springframework.boot.builder` package. The `showBanner(false)` method of this builder saves console space.

After a successful start, the Weather Microservice can be invoked. The final URL of a RESTful resource is assembled as `{protocol}://{host}:{port}/{base-context-path}/{resource-path}`. By default, a plain HTTP protocol without any SSL/TLS capabilities is used when Tomcat is started. As already mentioned, the default interface Tomcat, or any other servlet container available with Spring Boot, binds to is localhost, using port `8080`. The base context path is empty by default; however, in the Weather Microservice, a domain-descriptive prefix is bound using `'server.contextPath=/weather'` in the `application.properties` file. The `Temperature` resource is available under the `/temperature` context path, which is appended to `{base-context-path}`. As the average temperature endpoint resides on the `/temperature` address, there's nothing more to append to the final URL.

To invoke `WeatherMicroService`, specifically the average temperature endpoint, any tool with support for an HTTP protocol can be used. One such tool is cURL. The Average temperature endpoint is reachable from the preceding URL by making an HTTP request using the `GET` method. If there is no cURL tool available for your operating system, for this simple request, any web browser can be utilized:

```
> curl http://localhost:8080/weather/temperature
{"temperature":35.0,"temperatureScale":"CELSIUS"}
```

The operation is successful if Spring Framework replies with a `Temperature` object serialized as JSON, as demonstrated in the preceding code. There is also an option for Maven to start any Spring Boot Microservice. With the Spring Boot Maven plugin present in the project, Maven goals prefixed with Spring Boot can be used to start, stop, and repackage the application:

```
> mvn spring-boot:run
...several output omitted...
Tomcat started on port(s): 8080 (http)
Started WeatherServiceApplication in 2.305 seconds (JVM running for 2.601)
```

The Spring Boot Maven plugin executes the very same `java -jar` command already demonstrated, only the process is more automated.

Summary

Java is a rich platform and there are numerous ways to achieve the same goal. This introductory chapter presented an opinionated view on the process of creating a Microservice from scratch. As a first step, the very basic process of general Java project creation was explained. We saw that both Java EE and Spring grow from the same Java roots. In general, as a consequence of the richness of the Java ecosystem, even the project-creation phase can be performed in various ways, not to mention the steps of actually implementing the Microservice. The approaches shown in this chapter are time-proven, universal, and most likely to be applied on a real project.

The Weather Microservice implemented with Java EE demonstrates its long-term focus on convention over configuration. Besides the actual `Temperature` resource, represented by a single class complemented with very few JAX-RS metadata annotations, there it was only one value configured. The value is the context path of the REST API within the Microservice, nothing more. RESTful resources implemented with JAX-RS are automatically discovered and integrated with the rest of Java EE's functionality. After the implementation phase, it was shown how Maven, the project management tool, can be used to build a thin Java Web Archive. With the Weather Microservice built and packaged into a WAR, we saw how a Microservice can be deployed and run by a single command using a Payara Micro instance.

A Spring-Framework-based implementation of the Weather Microservice was performed on the same roots. Creating the actual resource required only one class, complemented with some metadata in the form of Spring REST MVC annotations. The next series of steps aimed to explain how modern Spring Framework applications are built, packaged, and run in an opinionated way named Spring Boot. First, we looked at how simple it is to configure Maven to produce a Spring Boot Java Archive. We emphasized how Spring Boot's JAR can be simply executed, as it contains all the dependencies required, including the parts of a servlet container. Finally, we saw how to implement a class that contains an entry point to the Spring Boot Microservice. We explained how the convention-over-configuration approach has reached the Spring world and which configuration steps are being done for the developer automatically.

Both Java EE and Spring Framework are very rich platforms, each with unique strengths. This chapter showed you a minimalistic use case for both platforms. The world of Microservices brings with it many new ideas and possibilities. In the chapters to come, such possibilities will be revealed and demonstrated. In the next chapter, we will discuss how various Microservices would interact with each other.

3
Connecting Microservices Together

In the previous chapter, we set up our development environment, chose Maven to create the skeleton of our project, and implemented a RESTful web service to provide average temperature information.

In this chapter, we'll start by creating a client that consumes the temperature service we created. We will be using a JAX-RS client API and Jersey open source reference implementation under the hood. We will use JAX-RS 2.1, which is a part of Java EE 8 and Jersey 2.26.

Jersey version 2.26 is the JAX-RS 2.1-compliant version and the latest available version at the time of writing.

It's also possible to use any JAX-RS-compliant implementations instead of Jersey, such as RESTEasy (`http://resteasy.jboss.org`) or Apache Wink (`https://wink.apache.org`).

We will also show how to implement the same logic for the client in an asynchronous way. We'll continue by implementing an interaction between two Microservices to demonstrate methods for integration. And finally, for discovering services, we'll register the services that we created into service discovery mechanisms that are available, such as Eureka and SnoopEE.

In this chapter, we are going to cover following topics:

- Building a client
- Connecting two Microservices

- Running Snoop and registering our service
- Installing and registering Eureka
- Consuming registered services
- Discovering and invoking the service

Building a client

To implement the client, we are going to create a standalone Java application that connects to our temperature service implemented in `Chapter 1`, *From Monoliths to Microservices*. We will consume the service with a JAX-RS-based client implementation. Let's start by defining the dependencies needed to implement this client application.

We are assuming that you've already created an empty Maven application of the `jar` packaging with your favorite IDE. You can also create the project with Maven Archetype, as shown in `Chapter 2`, *Creating Your First Microservice*. Since we are using Jersey as the reference implementation, its Maven dependency should be added, as follows:

```
<dependency>
   <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.26</version>
</dependency>
```

The Jersey client dependency transitively depends on `javax.ws.rs-api`, which is the dependency of the JAX-RS API version 2.1. We don't need to define the JAX-RS 2.1 dependency explicitly, since it will be resolved by Maven.

In order to use this dependency in a standalone console project, the Jersey-HK2 dependency should be also added, as follows:

```
<dependency>
    <groupId>org.glassfish.jersey.inject</groupId>
    <artifactId>jersey-hk2</artifactId>
    <version>2.26</version>
</dependency>
```

Jersey uses **HK2** (also known as **Hundred-Kilobyte Kernel**) by default for the dependency injection. You can find more details about HK2 at `https://javaee.github.io/hk2`.

The client implementation for consuming the `Temperature` service is implemented as follows:

```
public class TemperatureResourceClient {

    public static void main(String... args) {
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:8080/" +
                "weather-service/weather/temperature");

        Invocation.Builder builder = target.request();
        Temperature result = builder.get(Temperature.class);

        System.out.println(result);

        client.close();
    }
}
```

This is a console application that can be run directly without providing any parameters. Within the implementation, we first create a container instance of `javax.ws.rs.client.Client` by simply invoking the `newClient()` method on the `javax.ws.rs.client.ClientBuilder`;class.;`ClientBuilder` is the main entry point for using the client API, which itself is an abstract class. For `ClientBuilder` an implementation is to be provided. In our case, Jersey provides the implentation in the form of `JerseyClientBuilder`. Then we create an instance of `javax.ws.rs.client.WebTarget` by setting the target URL for the client. By calling the `request()` method on the target, we're starting to build a request for the targeted web resource. After having an instance of `javax.ws.rs.client.Invocation.Builder` at hand, calling the `get()` method with the `Temperature.class` parameter invokes the `HTTP GET` method on the URL synchronously. This is the default behavior of the client container, and invoking the `get()` method is a blocking call until the response has been sent by the remote endpoint.

We will be covering asynchronous client implementation in the following section. The parameter passed to the `get()` method states the class of the response type, which is an instance of the `Temperature` class in our case. As a final step, we close the client instance and all of its associated resources.

The aforementioned synchronous client implementation can also be implemented in a non-blocking way, as follows:

```
public class TemperatureResourceAsyncClient1 {

    public static void main(String... args)
            throws ExecutionException, InterruptedException {

        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:8080/" +
                "weather-service/weather/temperature");

        Invocation.Builder builder = target.request();
        Future<Temperature> futureResult = builder
                .async()
                .get(Temperature.class);
        System.out.println(futureResult.get());
        client.close();
    }
}
```

The preceding client implementation enables us to consume the REST endpoint asynchronously. The usage of the API is exactly the same as we did in the synchronous call, but we only specified the `async()` method call to the builder. As soon as we invoke the `get()` method, we use an asynchronous invoker, and the method will execute and return directly with an instance of `java.util.concurrent.Future`. So, in the next step, we are using that instance to actually retrieve the result or the async REST invocation.

One problem that can be faced when we invoke the `get()` method on `futureResult` is that if the method takes a long time to respond, our client code will be blocked. A timeout could be set on the `get()` method call, as follows:

```
try {
    System.out.println(futureResult.get(5, TimeUnit.SECONDS));
} catch (TimeoutException e) {
    // Handle timeout gracefully
}
```

The client application will not be blocked forever if a problem occurs on the server side. But in the worst-case scenario, the client will be blocked for five seconds at most. A better approach for implementing a non-blocking client would be using the `InvocationCallback` interface from JAX-RS 2.0, as follows:

```
public class TemperatureResourceAsyncClient3 {

    public static void main(String... args)
```

```
            throws ExecutionException, InterruptedException {

        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:8080/" +
                "weather-service/weather/temperature");
        Invocation.Builder builder = target.request();
        builder.async().
                get(new InvocationCallback<Temperature>() {
                    @Override
                    public void completed(Temperature t) {
                        System.out.println(t);
                    }
                    @Override
                    public void failed(Throwable throwable) {
                        // method that will be invoked
                        // when something goes wrong
                    }
                });
        client.close();
    }
}
```

The `InvocationCallback` interface provides two methods, `completed()` and `failed()`, which will be invoked when the call to the endpoint is completed or failed, respectively. Having reactive-style programming for this scenario will allow you to have a simpler implementation to surmount such programming difficulties. So JAX-RS 2.1 takes the client implementation one step further by introducing this reactive-style programming. Here is an example implementation of a reactive client approach:

```
public class TemperatureResourceAsyncClient4 {

    public static void main(String... args)
            throws ExecutionException, InterruptedException {
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:8080/" +
                "weather-service/weather/temperature");

        Invocation.Builder builder = target.request();
        CompletionStage<Response> response =
                builder.rx().get();

        response.thenAccept(res -> {
            Temperature t = res.readEntity(Temperature.class);
            System.out.println(t);
        });

        new Thread(() -> {
            try {
                for (int seconds = 3; seconds > 0; seconds--) {
                    System.out.println(seconds + " seconds left");
                    Thread.sleep(1000);
                }
                System.out.println("Finished!");
                client.close();
            }
            catch (Exception ignored) {}
        }).start();
    }
}
```

The `rx()` method invoked on the builder is introduced with JAX-RS 2.1, and its job is to get a reactive invoker that exists on the client's runtime. Since we are using Jersey on the classpath, an instance of `JerseyCompletionStageRxInvoker` will be returned in this implementation. Invoking the `get()` method on the reactive invoker will result in an instance of `CompletionStage<Response>`. `CompletionStage<T>` is a new interface introduced with Java 8, and it represents a computation stage that needs to be executed when another CompletionStage completes. Then we can invoke the `thenAccept()` method on the response to retrieve the result of the service call as an argument, named `res`.

We implemented a timer thread at the end of the client's main method to print out a countdown timer by starting from three seconds. So while counting down the seconds, we will also see the response of the service call printed out to the console. The whole output of the client would be as follows when executed:

```
3 seconds left
Temperature{temperature=35.0, temperatureScale=CELSIUS}
2 seconds left
1 seconds left
Finished
```

Using third-party reactive frameworks

It's also possible to use third-party reactive frameworks with JAX-RS to instantiate a reactive invoker. RxJava (`https://github.com/ReactiveX/RxJava`) is one one of the most advanced reactive libraries for Java, and Jersey provides support for it by offering a custom client artifact. We will implement a reactive client by using RxJava under the hood. The client's Maven dependency definition is as follows:

```
<dependency>
   <groupId>org.glassfish.jersey.ext.rx</groupId>
   <artifactId>jersey-rx-client-rxjava2</artifactId>
```

```
        <version>2.26</version>
    </dependency>
```

`jersey-rx-client-rxjava2` transitively depends on RxJava 2.0.4, so we don't need to define it in our dependency graph.

The following code gives implementation of `Temperature` Client:

```
public class TemperatureResourceAsyncClient5 {

    public static void main(String... args)
            throws ExecutionException, InterruptedException {

        Client client = ClientBuilder.newClient()
                .register(RxFlowableInvokerProvider.class);
        WebTarget target = client.target("http://localhost:8080/" +
                "weather-service/weather/temperature");

        Invocation.Builder builder = target.request();
        Flowable<Response> flowable = builder
                .rx(RxFlowableInvoker.class)
                .get();
        flowable.subscribe(res -> {
            Temperature t = res.readEntity(Temperature.class);
            System.out.println(t);
        });

        new Thread(() -> {
            try {
                for (int seconds = 3; seconds > 0; seconds--) {
                    System.out.println(seconds + " seconds left");
                    Thread.sleep(1000);
                }
                System.out.println("Finished!");
                client.close();
            }
            catch (Exception ignored) {}
        }).start();
    }
}
```

We first register a custom JAX-RS component, the RxFlowableInvokerProvider class, to our client container by calling the register() method. This integrates an instance of JerseyRxFlowableInvoker, which is an implementation of a reactive invoker specialized for io.reactivex.Flowable. Invoking the rx() method with the RxFlowableInvoker.class parameter returns the reactive invoker subclassed to RxFlowableInvoker, which in our case, is the same instance of JerseyRxFlowableInvoker.

Connecting two Microservices together

In Chapter 2, *Creating Your First Microservice*, where we implemented our first Microservice, we created a REST-based Microservice that returns artificial temperature information when requested. In this chapter, we are going to implement three core services, one will be the modified version of the temperature service where it gets a location as a path parameter and returns an artificial temperature related to that location. The other service will be the location service, which returns three defined locations. Our third service will be the forecast service, which fetches all locations from the location service we implemented and requests the temperature service for each location. All these calls will be done in synchronous mode.

The resource configuration for the application is as follows:

```
@ApplicationPath("/smartcity")
public class SmartCityServices extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<>();

        classes.add(LocationResource.class);
        classes.add(TemperatureResource.class);
        classes.add(ForecastResource.class);

        return classes;
    }
}
```

With the advent of new Java EE 8 specifications, the object of JSON mapping will be taken care of by JSON-B, the Java API for JSON Binding. The reference implementation of JSON-B is Yasson RI and it's being used implicitly. We are not using any implementations from Jackson, which is a well-known JSON library for Java, to handle the mapping.

You can find more details about JSON-B at `http://json-b.net` ; for more information about Yasson, check out `https://github.com/eclipse/yasson`.

The location service implementation is given as follows. It just returns three cities with their names defined inside instances of the `Location` class. The response is wrapped with `GenericEntity` to overcome the type-erasure problem, since the type will be removed at runtime, meaning that `List<Location>` will be set as `List<?>` .

The code below gives implementation of the Location service:

```
@Path("/location")
public class LocationResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getLocations() {

        List<Location> locations = new ArrayList<>();
        locations.add(new Location("London"));
        locations.add(new Location("Istanbul"));
        locations.add(new Location("Prague"));

        return Response
                .ok(new GenericEntity<List<Location>>(locations){})
                .build();
    }
}
```

The temperature service retrieves the city name as a path parameter, and randomly generates a temperature in degrees Celsius for the requested city. In order to simulate the temperature-measuring process, a delay of 500 milliseconds is added to the execution. Since we are doing service executions synchronously, the delay will accumulate for the forecast service. We will discuss asynchronous methods of communication among services in `Chapter 4`, *Asynchronous communication in Microservices.*

The following code showcases the delay in response:

```
@Path("/temperature")
public class TemperatureResource {

    @GET
    @Path("/{city}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAverageTemperature(
            @PathParam("city") String cityName) {

        Temperature temperature = new Temperature();
        temperature.setTemperature(calculateTemperature());
        temperature.setTemperatureScale(TemperatureScale.CELSIUS);

        try {
            Thread.sleep(500);
        } catch (InterruptedException ignored) {}

        return Response.ok(temperature).build();
    }

    private Double calculateTemperature() {
        Random r = new Random();
        int low = 30, high = 50;
        return (double) (r.nextInt(high - low) + low);
    }
}
```

The Forecast service that bundles both the location and temperature services is given in the preceding code snippet. We define the web targets for both the location and temperature services with `@Uri`, and then within the `getLocationsWithTemperature()` method. First, we invoke the location service, and then for each location that we have, we request the temperature service with that location.

The following code shows `getLocationWithTemprature` method implementation:

```
@Path("/forecast")
public class ForecastResource {

    @Uri("location")
    private WebTarget locationTarget;

    @Uri("temperature/{city}")
    private WebTarget temperatureTarget;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getLocationsWithTemperature() {
        long startTime = System.currentTimeMillis();
        ServiceResponse response = new ServiceResponse();

        List<Location> locations = locationTarget.request()
                .get(new GenericType<List<Location>>() {});

        locations.forEach(location -> {
            Temperature temperature = temperatureTarget
                    .resolveTemplate("city", location.getName())
                    .request()
                    .get(Temperature.class);

            response.getForecasts()
                    .add(new Forecast(location, temperature));
        });
        long endTime = System.currentTimeMillis();
        response.setProcessingTime(endTime - startTime);

        return Response.ok(response).build();
    }
}
```

After deploying the artifact onto Payara Micro, requesting the `http://localhost:8080/forecast-service/smartcity/forecast` URL will result in the following excerpt. Keep in mind that temperature values are randomly generated. So in our sample response, the processing time is 1,542 milliseconds, which is a total of 3 consecutive calls to the temperature service.

The following JSON shows the expected response:

```
    {
     "processingTime":1542,
     "forecasts":[
       {
          "location":{
             "name":"London"
          },
          "temperature":{
             "temperature":30.0,
             "temperatureScale":"CELSIUS"
          }
       },
       {
          "location":{
             "name":"Istanbul"
          },
          "temperature":{
             "temperature":30.0,
             "temperatureScale":"CELSIUS"
          }
       },
       {
          "location":{
             "name":"Prague"
```

```
        },
        "temperature":{
           "temperature":30.0,
           "temperatureScale":"CELSIUS"
        }
     }
  ]
}
```

When we implement this in an asynchronous way in `Chapter 4`, *Asynchronous Communication for Microservices,* the processing time will reduce drastically.

Creating and pooling web targets

JAX-RS provides a Client API for creating clients to connect to the services with its fluent API approach. The instances of `Client` are expensive to initialize, so there should be a minimum set of it for reuse. `WebTarget` represents the specific URI that will be invoked so it can easily be set into the instance of the `Client` before requesting. For the Forecast service given in the previous section, we can implement a web target producer to create the location web target when requested.

The following code showcases the implementation of `produceLocationWebTarget` :

```
@ApplicationScoped
public class WebTargetProducer {

    private Client client;

    @PostConstruct
    private void postConstruct() {
        this.client = ClientBuilder.newClient();
    }

    @Produces
    @Dependent
    public WebTarget produceLocationWebTarget() {
        return client.target("http://localhost:8080")
                .path("forecast-service")
                .path("smartcity")
                .path("location");
    }
}
```

And the web target can be produced within the Forecast service, as follows:

```
List<Location> locations = producer.produceLocationWebTarget()
    .request()
    .get(new GenericType<List<Location>>() {});
```

Making Microservices discoverable

Within a monolith application, services/code parts call each other through method calls at the implementation level. In a distributed environment, this won't be the case, since services will be running in a well-known location, and that location would be pointed out with a hostname and related port information within an HTTP-based implementation. Things will get more complicated in a Microservices-based architecture, where services can be virtualized or containerized. This would lead to an even more dynamic environment where the number of service instances and their deploy locations change frequently. To overcome this problem by addressing these insufficiencies, we will introduce the concept of service discovery. Service discovery is a mechanism that allows other execution blocks to discover information about our registered service.

We will demonstrate the Service discovery mechanisms by employing two frameworks: Snoop and Eureka. Snoop will be used for Java-EE-based Microservices, and Eureka will be used for Spring-based Microservices.

Snoop

Snoop is a framework that provides a service registration and lookup mechanism for Microservices based on Java EE. We will register our existing temperature service to Snoop and consume it via a Snoop-powered client. Snoop currently provides two versions: 1.x and 2.x. The 2.x version is a work in progress, so we will continue with the latest 1.x release that is available.

Running Snoop and registering our service

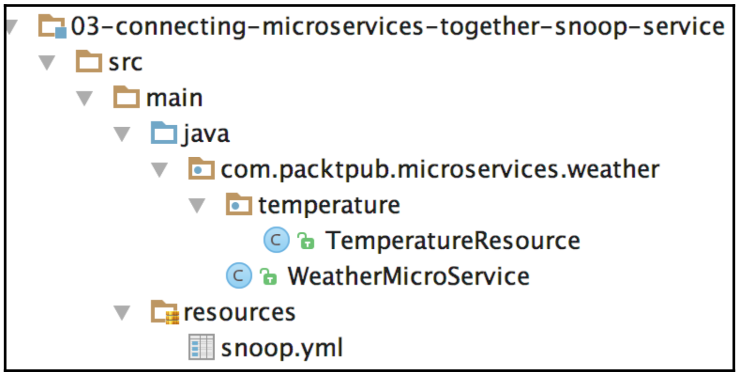
First, we need to download Snoop Service, which is available as a `.war` file, and deploy it onto Payara Micro with our sample service, packaged as `.war`.

The latest version of Snoop Service is 1.3.4 and it available to download from `http://central.maven.org/maven2/eu/agilejava/snoop-service/1.3.4/snoop-service-1.3.4.war`.

Then, we are going to implement a modified version of our Temperature service. We'll start by adding the Snoop artifact dependency, as follows:

```
<dependency>
    <groupId>eu.agilejava</groupId>
    <artifactId>snoop</artifactId>
    <version>1.3.4</version>
</dependency>
```

The project structure for the Temperature service is as follows. We'll have an application configuration class, `WeatherMicroService`, a service implementation ;`TemperatureResource`, and a Snoop configuration file, `snoop.yml`:



```
03-connecting-microservices-together-snoop-service
    src
        main
            java
                com.packtpub.microservices.weather
                    temperature
                        TemperatureResource
                    WeatherMicroService
            resources
                snoop.yml
```

The application configuration of the deployed service will be changed a little bit to contain the Snoop-specific `@EnableSnoopClient` annotation. This annotation enables a web application to act as a Snoop client and it helps to register our application as a service under the Snoop Service Registry. `serviceName` is the unique identifier for this service when it is registered with Snoop Service:

```
@EnableSnoopClient(serviceName = "weatherSnoop")
@ApplicationPath("/weather")
public class WeatherMicroService extends Application {
}
```

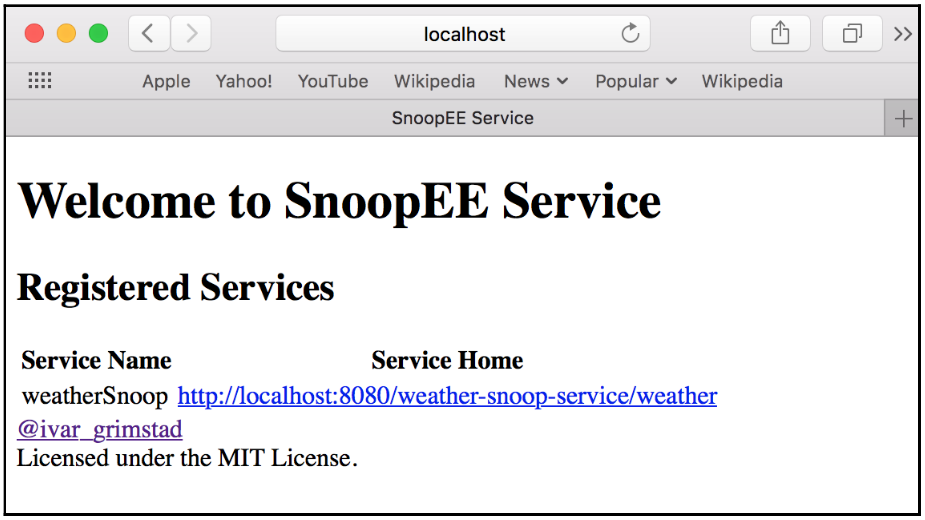
The `TemperatureResource` class implementation would be the same as we had in the previous chapter, `Chapter 2`, *Creating your first Microservice*. The Snoop configuration can either be done in the following order, with the priority of system properties, environment variables, or a provided `.YML` file. We used the following `.YML` file as our service registration configuration:

```
snoop:
 host: http://localhost
 port: 8080
 serviceRoot: weather-snoop-service/weather
 snoopService: localhost:8080/snoop-service/
```

After wrapping up the implementation, it's time to deploy it onto Payara Micro, along with the Snoop Service:

```
java -jar /path/to/payara-micro.jar --deploy /path/to/weather-snoop-
service.war --deploy /path/to/snoop-service-1.3.4.war
```

After deployment, the URL for the dashboard of Snoop service will be `http://localhost:8080/snoop-service`, which lists all registered services. As you can see, our `weatherSnoop` service got registered with its home URL:



Welcome to SnoopEE Service

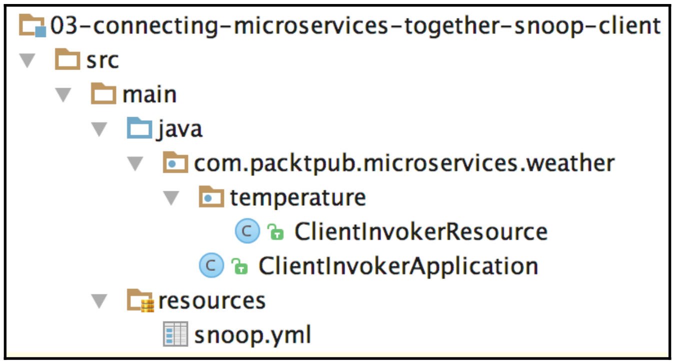
Registered Services

Service Name	Service Home
weatherSnoop	http://localhost:8080/weather-snoop-service/weather

@ivar_grimstad
Licensed under the MIT License.

Consuming registered services

Now, it's time to consume our registered service. To achieve this, we implement another REST service, which acts as a client. The project structure for the client is as follows:



```
03-connecting-microservices-together-snoop-client
  src
    main
      java
        com.packtpub.microservices.weather
          temperature
            ClientInvokerResource
          ClientInvokerApplication
      resources
        snoop.yml
```

`ClientInvokerResource` is the actual service, with the implementation given in the following code block. We are injecting a Snoop-based client with the help of the `@Snoop` annotation, and the `serviceName` attribute matches the name that we used when registering the service:

```
@Path("/invoke")
@RequestScoped
public class ClientInvokerResource {
    @Inject
    @Snoop(serviceName = "weatherSnoop")
    private SnoopServiceClient client;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public String invoke() {
       String response = client.simpleGet("temperature")
          .filter(r -> r.getStatus() == 200)
          .map(r -> r.readEntity(String.class))
          .orElse("Problem occurred!");
       return response;
   }
}
```

The following command sh0ws deploying the application onto Payara micro with port auto-binding enabled:

```
java -jar /path/to/payara-micro.jar --deploy /path/to/weather-snoop-
client.war --autoBindHttp
```

We can use the URL `http://localhost:8081/weather-snoop-client-1.0/client/invoke` and it gives us the response from the Temperature service we deployed before.

SnoopEE is not a load-balancer, so if you are running multiple instances of the same Microservice, it should be load balanced with a load-balancer framework.

Eureka

Eureka is a service from Netflix (`https://netflix.github.io`) that provides features for locating services to handle load balancing and failover mechanisms. Eureka is being actively developed in two separate branches: version 1.x and 2.x. The 2.x version is a work in progress, and it's an evolution of the 1.x version. We'll stick with the 1.x version for its maturity, wide usage, and integration with Spring Cloud services.

Installing Eureka Server

First, we're going to create the server project to initialize the Eureka Server by adding the Spring Cloud dependency, as follows:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
    <version>1.4.0.RELEASE</version>
</dependency>
```

At the time of writing, the latest available version of Spring Cloud services is 1.4.0.RELEASE.

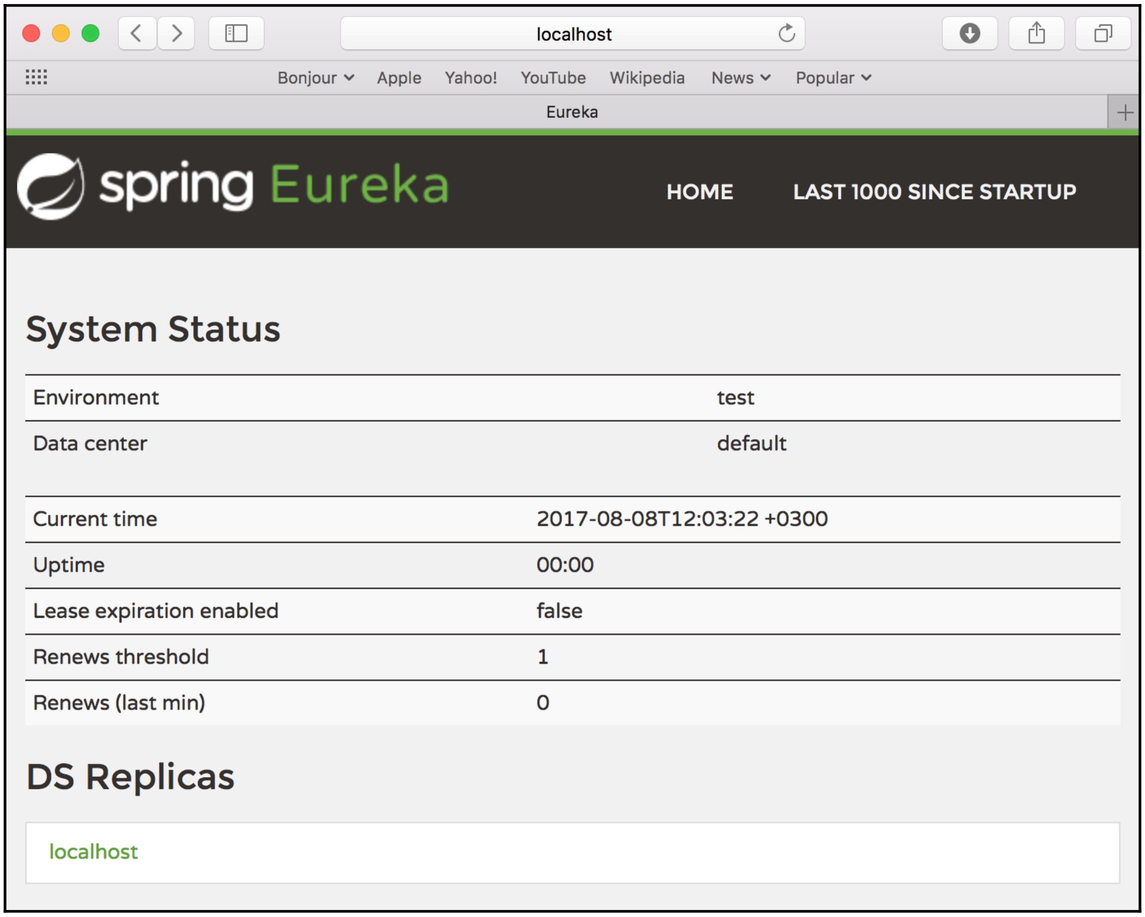
Within the code of a Spring Boot application, we are going to enable the Eureka Server related configuration by adding the `@EnableEurekaServer` annotation:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaWeatherServiceServer {
    public static void main(String... args) {
        SpringApplication.run(EurekaWeatherServiceServer.class);
    }
}
```

The `application.properties` file for configuring the Eureka Server is as shown in the following code. `server.port` defines the port that will be used by the server. `8761` is the most commonly used port number used for the server; if it's not provided, the Server will allocate `8080` by default. With `eureka.client.register-with-eureka`, we are telling the Eureka built-in client not to register itself since our application should be acting as a server:

```
server.port=8761
eureka.client.register-with-eureka=false
```

After invoking the `main` method on the `EurekaWeatherServiceServer` class, and then requesting `http://localhost:8761`, you should see the dashboard of Eureka Server, as follows:



System Status

Environment	test
Data center	default

Current time	2017-08-08T12:03:22 +0300
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

localhost

Registering the service

Next, we are going to reuse the Spring Boot-based temperature service that we created in `Chapter 1`, *From Monoliths to Microservices*.

First, we'll define the Maven dependencies for Eureka as shown here:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    <version>1.4.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.0</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-eureka-client</artifactId>
    <version>1.4.0.RELEASE</version>
</dependency>
```

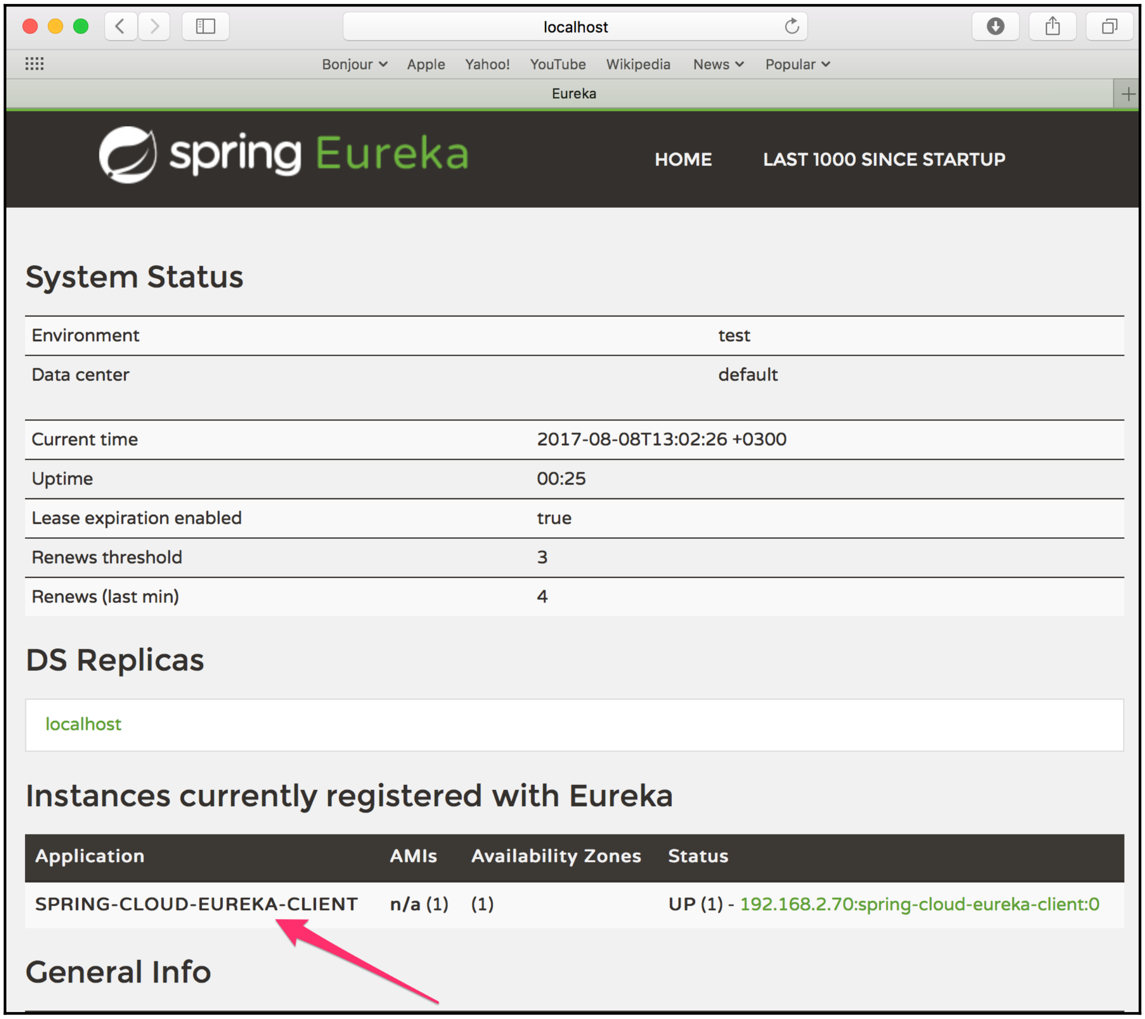
Then, we'll add the `@EnableEurekaClient` annotation to our application configuration:

```
@EnableEurekaClient
@SpringBootApplication
public class EurekaWeatherService {
    public static void main(String... args) {
        SpringApplication.run(EurekaWeatherService.class);
    }
}
```

To make our temperature service aware of the Eureka Server, we are going to add some configurations, defined in a `.YML` file, as follows:

```
spring:
   application:
      name: spring-cloud-eureka-client
server:
   port: 0
eureka:
   client:
      serviceUrl:
         defaultZone: ${EUREKA_URI:http://localhost:8761/eureka}
    instance:
         preferIpAddress: true
```

After invoking the `main` method on the `EurekaWeatherService` class and then requesting `http://localhost:8761`, you should see that our application is registered with the server:



System Status

Environment	test
Data center	default

Current time	2017-08-08T13:02:26 +0300
Uptime	00:25
Lease expiration enabled	true
Renews threshold	3
Renews (last min)	4

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
SPRING-CLOUD-EUREKA-CLIENT	n/a (1)	(1)	UP (1) - 192.168.2.70:spring-cloud-eureka-client:0

General Info

Discovering and invoking the service

Now it's time to consume our registered service. We'll discover it and then consume it with Spring's `RestTemplate` by providing the host and port values of the service:

```
@EnableEurekaClient
@SpringBootApplication
public class EurekaWeatherServiceClient {
    public static void main(String... args) {
        SpringApplication.run(EurekaWeatherServiceClient.class);
    }
    @Autowired
    private EurekaClient eurekaClient;
    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }

    @Bean
    public CommandLineRunner run(RestTemplate restTemplate)
            throws Exception {
        Application application = eurekaClient
                .getApplication("spring-cloud-eureka-client");
        InstanceInfo instanceInfo = application.getInstances().get(0);
        String hostname = instanceInfo.getHostName();
        int port = instanceInfo.getPort();

        return args -> {
            String result = restTemplate.getForObject(
              "http://" + hostname +
              ":" + port + "/temperature", String.class);
            System.out.println(result);
        };
    }
}
```

Summary

In this chapter, we created a client to consume a service by using Jersey. We also showed how the client could be implemented asynchronously. Then, we introduced third-party reactive frameworks, and detailed how they integrate with the latest version of JAX-RS.

To connect Microservices together, we created a Forecast service that consumes our location and Temperature services under the hood. The Forecast service synchronously consumes the location service, and for each location it returns, it invokes the Temperature service to fetch the temperature of that location. The next chapter will introduce asynchronous approaches for connecting the Microservices.

To demonstrate service discovery mechanisms, we looked at the Snoop and Eureka frameworks by registering Java-EE-based and Spring-based Microservices. Then we demonstrated how services can be discovered and invoked by implementing client code.

In the next chapter, we will show you ways to integrate asynchronous communication mechanisms between our demonstrated services, ranging from simple scenarios to more complex ones.

4
Asynchronous Communication for Microservices

In the previous chapter, we created a Forecast service that consumes Location and Temperature services to produce results showcasing the weather forecast. We also explored ways to register and discover services that are available on the network. All of that is very easy and straightforward. Very often, however, you'll need to optimize the communication between services, which is where asynchronous communication can be helpful.

In this chapter, we'll discuss how we can introduce asynchronous processing and communication to our services. There are many levels of asynchronous communication, from simpler ones to more robust and complex ones. Complexity has its own costs, therefore we'll start by adding asynchrony to the communication in a couple of small steps.

We'll modify the Forecast REST service to call other services more effectively, calling them in parallel and receiving their responses asynchronously when they are ready. Then, we'll modify it further to return a stream of data via the **Server-Sent Events** (**SSE**) protocol so that the data can be processed immediately when available.

We will discuss WebSocket-based communication and look into the details of message-oriented and queue-based communication. We will also cover point-to-point and publisher-subscriber patterns of communication.

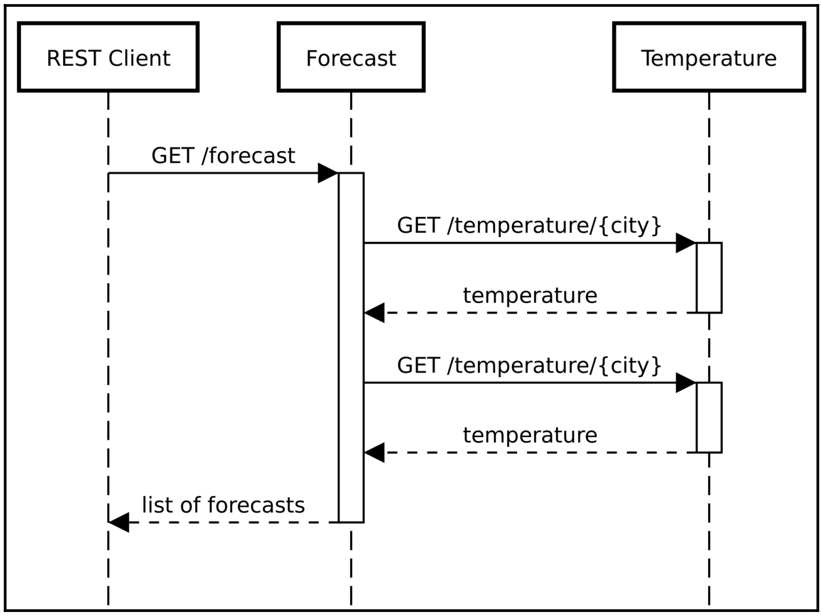
In a nutshell, the following topics will be covered in this chapter:

- Speeding up services with a Reactive API
- Streaming responses with SSE
- Two-Way Asynchronous services with WebSocket
- Decoupling services with Message-Oriented Middleware

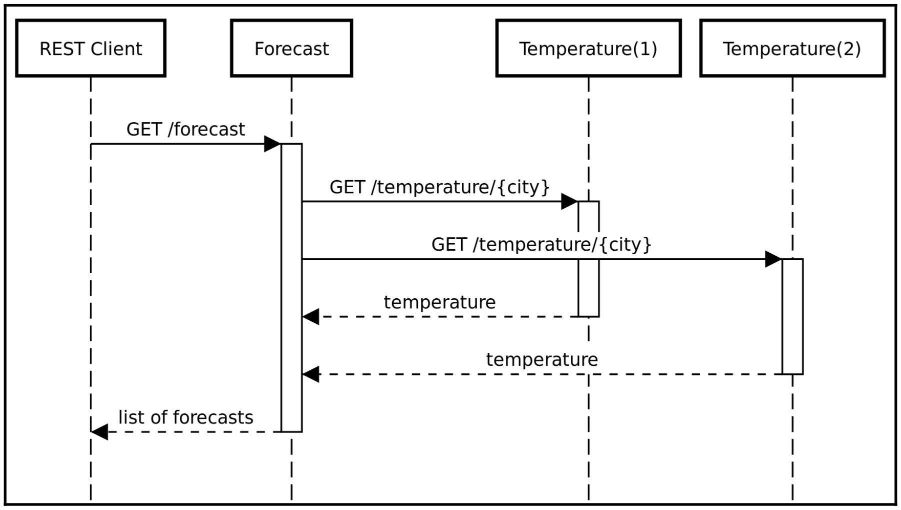
Speeding up services with the Reactive API

Services can often be optimized with asynchronous processing, even without changing their behavior toward the outside world. The reason why some services aren't efficient is that they need to wait for other services to provide a result to continue further. This is especially true when a service executes several calls to other services that don't have to wait for each other. Since the calls are independent, they can be executed in parallel and produce results much faster than if called sequentially.

Looking at the Forecast service in detail, we'll find out that it calls the Temperature service multiple times sequentially in a loop for each city. It waits for each call to finish and return results before doing another call. The following diagram illustrates what's happening:



In order to be more efficient, the Forecast service could do all the calls to the Temperature service in parallel. All the calls are completely independent of each other so there's no need to wait for one to return results before doing another call. The following diagram shows our goal:



Since the calls to the Temperature service happen simultaneously, they can finish much faster than in the sequential scenario. The Forecast service can, therefore, produce the final list of forecasts in a shorter time.

The *Building a client* section in `Chapter 3`, *Connecting Microservices Together*, explained how to use the JAX-RS request builder's `async()` or `rx()` methods to retrieve the results of a single asynchronous call. We now need to compose the results of multiple asynchronous calls. Therefore, we will use the `rx()` method, which provides an easy way to do it with a convenient, fluent API.

Collecting results of multiple parallel calls

We will use a combination of Java streams and CompletionStage to build a pipeline that processes the results as they arrive, and merges them into a single response. The first part of the pipeline will call the Temperature service for each location in parallel and specify what to do with each received result. Using the `rx()` method on the temperature request builder allows us to call a version of the `get` method, which immediately returns a `CompletionStage` result. We can then chain a Lambda expression that converts a received temperature into a forecast when the result of the call arrives, as in the following code snippet:

```
CompletionStage<Forecast> forecastStage = request.rx()
        .get(Temperature.class)
        .thenApply(temperature -> new Forecast(location, temperature));
```

Note that the value in `forecastStage` is a `CompletionStage` of `Forecast`, and not a `Forecast` itself. It is used to build the asynchronous pipeline, while the value of the forecast will be available to other chained lambda expressions as their argument when it's available.

The complete first part of the pipeline is implemented as follows:

```
CompletionStage<List<Forecast>> initialStage
    = CompletableFuture.completedFuture(new ArrayList<>());
CompletionStage<List<Forecast>> finalStage = locations.stream()
    .map(location -> {
        return temperatureTarget
            .resolveTemplate("city", location.getName())
            .request().rx().get(Temperature.class)
            .thenApply(temperature -> new Forecast(location, temperature));
    })
    .reduce(initialStage, (combinedStage, forecastStage) -> {
            return combinedStage.thenCombine(forecastStage, (forecasts,
forecast) -> {
                forecasts.add(forecast);
                return forecasts;
            });
        },
        (stage1, stage2) -> null); // combiner won't be used, return null
```

Since we need to execute multiple calls in parallel, we execute single asynchronous calls in a stream of locations to process all locations and their temperatures. Running the asynchronous `rx().get()` method on a stream of locations executes the calls to the Temperature service in parallel. Each call to the Temperature service is followed by building an asynchronous pipeline for each returned `CompletionStage`. An asynchronous pipeline is a chain of lambdas that is automatically executed on a different thread later, when the result of an asynchronous call is available. In the end, we reduce a stream of `CompletionStage` results into a single `CompletionStage` that is completed after all individual asynchronous executions are complete. This is to synchronize the parallel execution and merge the results into a list of forecasts.

Completing a REST response asynchronously

After all parallel executions are synchronized to a single stage, and the results combined into a list, we can chain more lambda callbacks to process the final list of forecasts and send it as a REST response. Because asynchronous callbacks are executed in a different thread, we need a way to finalize the REST response other than returning a final value from the REST method. Although we could put the initial thread to sleep and wait until the asynchronous result is available, it has serious disadvantages. One of them is that we need two threads to finalize the response, while only one of them is working at a given time. Another problem is that we need to synchronize the threads and pass the list of forecasts from one to the other. Synchronization between threads is an additional overhead, which slows things down and should be avoided if possible. A better solution is to return from the initial method and complete the response later in another thread. This is called asynchronous request processing.

The JAX-RS REST API supports asynchronous processing and allows us to provide a response even after a resource method returns. If we return `CompletionStage` directly from a resource method, the container will understand it, complete it, and send the HTTP response after the returned `CompletionStage` completes.

`CompletionStage` as a return value in resource methods has only been supported since Java EE 8. JAX-RS also supports a lower-level way of returning responses asynchronously, using an `AsyncResponse` argument in resource methods, which can also be used in older versions of Java EE.

Therefore, to make our method asynchronous, we just replace the `Response` return type with `CompletionStage<Response>`:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public CompletionStage<Response> getLocationsWithTemperature()
```

Our returned `CompletionStage` can hold any type that's allowed as a return value of a synchronous method. It can hold any Java type that can be mapped to a REST response. Therefore, instead of `Response`, we will use a custom `ServiceResponse` class, which will be automatically mapped to JSON using the JSON-Binding mechanism available in Java EE 8:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public CompletionStage<ServiceResponse> getLocationsWithTemperature()
```

Lastly, we need to return the final stage of our asynchronous computation. In our previous example, this is stored in the `finalStage` variable:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public CompletionStage<ServiceResponse> getLocationsWithTemperature() {
   ...
   return finalStage;
}
```

In our case, the type of the `finalStage` variable is different than the type of `CompletionStage` returned from the method. Therefore, we're going to convert `finalStage` with its `apply` method:

```
return finalStage.thenApply(forecasts -> {
           ServiceResponse response = new ServiceResponse();
           response.getForecasts().addAll(forecasts);
           return response;
       });
```

We closed the loop by finishing our asynchronous callback chain and returning the final CompletionStage from the REST resource method. The container does the rest; it waits for `CompletionStage` to complete and either sends an expected response when it's completed (OK), or it sends an error response and logs the error when it's completed with an exception.

As an alternative to returning `CompletionStage`, we can use a supplied `AsyncResponse` object. This is useful if we need to access the JAX-RS API that isn't available with plain `CompletionStage`. It may also be convenient if we don't use `CompletionStage` during our processing (at all), and we don't want to create an instance of `CompletionStage` (for example, `CompletableFuture`) just so that we can return it.

An alternative asynchronous definition of our `getLocationsWithTemperature` method with an `AsyncResponse` argument looks as follows:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public void getLocationsWithTemperature(@Suspended AsyncResponse ar) {
}
```

This asynchronous method should do the following:

- Return no value (the return type is `void`)
- Contain an argument of the `javax.ws.rs.container.AsyncResponse` type
- Make sure that the `AsyncResponse` argument is annotated with `javax.ws.rs.container.Suspended`

The method doesn't return anything immediately, hence the calling method is not blocked and it continues with other commands. The container understands that the method is asynchronous and that a response will be provided via the `AsyncResponse` object later, from another thread. In our case, the response will be provided from a thread that is created for us automatically to handle asynchronous responses from remote REST services.

It's possible to complete the response using `AsyncResponse` within the method call on the same thread before the method returns. In this case, the container will complete the response right after the resource method returns. However, the method then behaves like a usual synchronous method and there's no advantage to using the asynchronous API.

When the final result is ready, we complete the response by calling the `resume` method of our `AsyncResponse` object. This method accepts the same values, which can be returned from a usual synchronous REST method. Therefore, we can call it with the same value, as in our previous synchronous version of the Forecast service:

```
response.getForecasts().addAll(forecasts);
asyncResponse.resume(Response.ok(response).build());
```

To summarize, asynchronous methods are supposed to return quickly, to allow further execution on the current thread. Since I/O operations and some other actions may take a while, the result may not be available without waiting. The result is available later and very often in another thread. Therefore, asynchronous methods can't return the result directly as the return value. They either don't return any value and provide the result using an additional **argument** (**callback**), or they return an **object** (**future** or **promise**) that will allow us to handle the result when it's available in the future.

An asynchronous JAX-RS method can have both forms. It either doesn't return any value and a final result or exception is provided by methods of the `AsyncResponse` argument, which is annotated with `@Suspended` to turn on asynchronous processing. This is suitable if we need more control over when and how we complete processing a request with a result or an exception, or if we need to specify a timeout or register life cycle callbacks. Alternatively, an asynchronous method simply returns `CompletionStage`, which either completes with a response or results in an exception.

Asynchronous exceptions

We're almost done, but we need to cover two important things that we have to remember when writing asynchronous code: **exceptions** and **timeouts**. In synchronous code, we catch exceptions by using a `try..catch` block. Timeouts also trigger exceptions, so we can handle them in the same way. With asynchronous invocations, we catch exceptions in a different way. Exceptions are handled in callbacks, in a similar way to how we handle a future return value if the execution completes normally. The `CompletionStage` interface provides methods to register exception callbacks. Unless we return `CompletionStage` from a method, we have to remember to register an exception callback to handle exceptions, otherwise they would be silently ignored.

This is very different from a synchronous execution, where exceptions are propagated out of a method and are either caught in another component, by a container, or by the JVM as a last resort. In the asynchronous world, if there's no callback to handle an exception, we risk that nothing else will handle it and we won't find out about it at all. Ignoring exceptions may lead to lots of negative consequences, so we should always remember to add an exception callback at the end of asynchronous execution.

The simplest way of catching exceptions with `CompletionStage` in an asynchronous JAX-RS method is to return `CompletionStage`. In this case, the container will register an exception handler and properly handle the exception, usually by printing an error message into the log. If we want to handle the exception ourselves, we can provide an exception handler with the `exceptionally` method. This method expects a lambda function that accepts a `Throwable` argument. The lambda function is executed if any of the previous stages ends with an exception, skipping all the intermediate stages. We will use this method to complete the REST response with an exception by calling the `resume` method with an exception instead of a result value, as follows:

```
finalStage.exceptionally(e -> {
    Throwable cause = (e instanceof CompletionException) ? e.getCause() :
e;
    asyncResponse.resume(cause);
    return null;
});
```

Because the `exceptionally` method almost always wraps the original exception into `CompletionException`, we retrieve the original exception with the `getCause` method. We also return the `null` value, just to satisfy the contract of the callback, as no further stage will use that value.

Specifying and handling asynchronous timeouts

In synchronous execution, default timeouts usually exist to avoid blocking a thread eternally. Even if there's no timeout, there's always a thread waiting for a result or an exception that indicates something is going on. This is not the case in asynchronous execution because no thread is waiting. If no callback or exception is triggered, there's nothing to indicate that a request is still in progress. Adding a timeout is essential to ensure a request is completed at some point, and we don't lose track of it.

In an asynchronous JAX-RS method, setting a timeout is as easy as calling the `setTimeout` method on the `AsyncResponse` object. We call this method as soon as possible at the beginning of an asynchronous method to start the timeout scheduler. If the specified time elapses since the `setTimeout` method was called, the REST response is completed with an exception even without calling the `resume` method, as follows:

```
public void getLocationsWithTemperature(@Suspended AsyncResponse
asyncResponse) {
    asyncResponse.setTimeout(5, TimeUnit.MINUTES);
    ...
}
```

A complete reactive REST service

Here's a complete method in the `ForecastResource.java` file:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public void getLocationsWithTemperature(@Suspended AsyncResponse
asyncResponse) {
    asyncResponse.setTimeout(5, TimeUnit.MINUTES);

// initial completed stage to reduce all stages into one
    CompletionStage<List<Forecast>> initialStage
        = CompletableFuture.completedFuture(new ArrayList<>());
    CompletionStage<List<Forecast>> finalStage = locations.stream()
// for each location, call a service and return a CompletionStage
        .map(location -> {
            return temperatureTarget
                .resolveTemplate("city", location.getName())
                .request()
                .rx()
                .get(Temperature.class)
                .thenApply(temperature -> new Forecast(location,
temperature));
```

```
        })
// reduce stages using thenCombine, which joins 2 stages into 1
        .reduce(initialStage,
            (combinedStage, forecastStage) -> {
                return combinedStage.thenCombine(forecastStage,
                    (forecasts, forecast) -> {
                        forecasts.add(forecast);
                        return forecasts;
                    });
            }, (stage1, stage2) -> null); // a combiner is not needed

// complete the response with forecasts
    finalStage.thenAccept(forecasts -> {
            asyncResponse.resume(Response.ok(forecasts).build());
        })
// handle an exception and complete the response with it
        .exceptionally(e -> {
// unwrap the real exception if wrapped in CompletionException)
            Throwable cause = (e instanceof CompletionException) ?
e.getCause() : e;
            asyncResponse.resume(cause);
            return null;
        });
}
```

Simplifying the code with a third-party reactive framework

Although we've improved the performance of our service, you must have noticed that the code became a lot more complex. We can simplify it by using a framework that provides an alternative API to `CompletionStage` for chaining asynchronous callbacks and joining results of parallel executions.

In `Chapter 3`, *Connecting Microservices Together*, you learned how to use RxJava with a JAX-RS client. We're going to use RxJava now, to replace `CompletionStage` with `Flowable`, and simplify our code.

The first part, which calls remote REST services in parallel, will be very similar to what we already have. The difference is mostly in using `RxFlowableInvoker` to retrieve a stream of `Flowable` instead of `CompletionStage`. We don't need to reduce the stream into a single `Flowable`; because RxJava provides a much more convenient way to join `Flowable`:

```
Stream<Flowable<Forecast>> futureForecasts = locations.stream()
    .map(location -> {
        return temperatureTarget
            .register(RxFlowableInvokerProvider.class)
            .resolveTemplate("city", location.getName())
            .request()
            .rx(RxFlowableInvoker.class)
            .get(Temperature.class)
            .map(temperature -> new Forecast(location, temperature));
    });
```

In the second part, we use the static `concat` method on the `Flowable` class to join all flowables into a single one:

```
Iterable<Flowable<Forecast>> iFutureForecasts = futureForecasts::iterator;
Flowable.concat(iFutureForecasts)
    .doOnNext(forecast -> {
        response.getForecasts().add(forecast);
    })
    .doOnComplete(() -> {
        asyncResponse.resume(Response.ok(response).build());
    })
    .doOnError(asyncResponse::resume)
    .subscribe();
    }
```

Unlike the final `CompletionStage` we had before, which worked with the final list of results, the `Flowable` we get now is a stream that produces multiple values. We handle each of them with the `doOnNext` method. A similar method wasn't available with `CompletionStage`, and that's why we had to concatenate all results in a list in a single `CompletionStage` instance.

Using RxJava and `Flowable` enables us to process results of multiple parallel calls as a stream of results, immediately after individual results are available. This makes the code more efficient and is simpler than reducing all results into a `CompletionStage`.

Another new thing is that we call `subscribe` after we build the execution pipeline. This is because a call with `RxFlowableInvoker` is executed lazily only after we declare interest in the values of the returned `Flowable`, for example, by calling the `subscribe` method.

Finally, similarly to using `CompletionStage`, we complete the response within the `doOnComplete` method and handle exceptions within the `doOnError` method.

Streaming responses with SSE

In the previous chapters, you learned how to create a REST service that returns a complete response at once. Most REST services work like this because of the nature of the underlying HTTP protocol. In this chapter, you've learned how to optimize the execution of such a REST service if it has to wait for other services. However, even optimized services may become slow to respond, because they can't be faster than the slowest call to another service. When completing a response takes a while, the client has to wait until all the data is prepared and the response is finally sent. This may have a range of negative consequences, which we can avoid if we send the data as soon as it's available in a stream of messages. The most natural way to do it in a REST service is using SSE, which works over the same HTTP protocol.

Building an asynchronous service

SSE is a web standard (`https://www.w3.org/TR/eventsource/`) that specifies how web servers can send a stream of messages during a potentially unlimited period of time. The connection has to be initiated by a client, and then the client only listens for messages received from the server. It works very similarly to a standard HTTP request, with the exception that the response can be sent in multiple chunks.

In `Chapter 3`, *Connecting Microservices Together*, we built a Forecast service, which retrieves data from location and temperature services and sends a list of forecasts in the response. Now, we're going to implement the same Forecast service as an asynchronous service. We will be providing forecasts as messages using SSE instead of sending all as a single response. Individual forecasts will be sent immediately when retrieved from the `Temperature` service, without waiting for other forecasts to be ready.

The same JAX-RS mechanism we used in `Chapter 3`, Connecting Microservices Together, to build simple REST services also supports building SSE server endpoints. Therefore, we can reuse a big portion of the code we built in the `Chapter 3`, *Connecting Microservices Together*.

The support for SSE is completely new in Java EE 8. The JAX-RS component added support for building SSE endpoints in the latest version, 2.1, which is part of Java EE 8.

We're going to start with the services we created in the `Chapter 3`, *Connecting Microservices Together*. We'll be using the same `LocationResource`, `TemperatureResource`, and `ForecastResource` classes. We'll need the same `Forecast`, `Temperature`, and `Location` data objects, which will be mapped to the JSON payload of the SSE messages. And we'll need the same configuration we've already used in the `SmartCityServices` class.

In order to make the Forecast service asynchronous, we're going to modify the `ForecastResource` class. Let's start with a skeleton of this class, which is already modified to send SSE events in the response.

This is the initial skeleton of the `ForecastResource` class:

```
@Path("/forecast")
public class ForecastResource {
   @Uri("location")
   private WebTarget locationTarget;

   @Uri("temperature/{city}")
   private WebTarget temperatureTarget;

   @GET
   @Produces(MediaType.SERVER_SENT_EVENTS)
   public void getLocationsWithTemperature(
      @Context SseEventSink eventSink,
      @Context Sse sse) {
         // ... method body
   }
}
```

The method body will do a very similar job as the simple synchronous `ForecastResource` class in `Chapter 3`, *Connecting Microservices Together*. The difference is that it will send the response using the injected `SseEventSink` instead of returning the response as a result of the method. We'll use the `send` method of the `SseEventSink` object to send the data. And we need to create a data object that holds the message to be sent. For that, we'll use a builder acquired from the second injected object of the `Sse` type.

Here's an example of sending a message in JSON format with a payload in a variable named response and with an ID set as `"event1"`:

```
OutboundSseEvent.Builder sseDataEventBuilder = sse.newEventBuilder()
   .mediaType(MediaType.APPLICATION_JSON_TYPE);
eventSink.send(
   sseDataEventBuilder
      .data(response)
```

```
            .id("event1")
            .build()
    );
```

TIP

Keep in mind that every method call on the `OutboundSseEvent.Builder` builder object modifies its internal structure. Make sure that you overwrite all previously set values if you don't want to set them for the new event, or use a new instance of the event builder.

All the steps in the new method body can be summarized into the following points:

- The list of locations is retrieved from the Location service as in the previous `Chapter 3`, *Connecting Microservices Together*.
- For every location, the forecast temperature will be retrieved using the Temperature service, again as in previous chapter.
- Every retrieved temperature will be sent as an event, using the `send` method of `SseEventSink`.
- After all the locations are processed, a final event with an ID of completed will be sent to indicate the end of the event stream.
- The SSE connection is closed by calling the close method of `SseEventSink`.

TIP

For a finite stream of events, we recommend sending a termination event after all data events are sent. This is to detect the end of the event stream on the client side safely and without any doubt. Some SSE clients don't detect the end of the stream correctly and try reconnecting infinitely after the connection is closed, assuming the connection was prematurely interrupted and new data is still to come.

And here is the complete source code of the `getLocationsWithTemperature` method, which executes the preceding points:

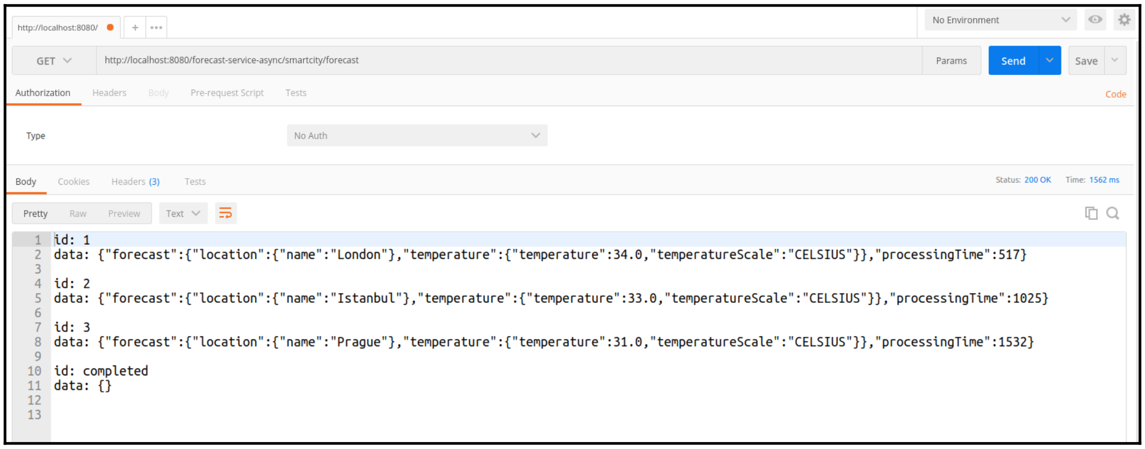
```
@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
public void getLocationsWithTemperature(
        @HeaderParam(HttpHeaders.LAST_EVENT_ID_HEADER) Integer lastEventId,
        @Context SseEventSink eventSink,
        @Context Sse sse) {
    OutboundSseEvent.Builder sseDataEventBuilder = sse.newEventBuilder()
            .mediaType(MediaType.APPLICATION_JSON_TYPE);
    List<Location> locations = locationTarget.request()
            .get(new GenericType<List<Location>>() {});
    int eventId = 0;
    for (Location location : locations) {
        eventId++;
        if (lastEventId != null && eventId <= lastEventId) {
            continue;  // skip to the last ID before reconnection
        }
        Temperature temperature = temperatureTarget
            .resolveTemplate("city", location.getName())
            .request()
            .get(Temperature.class);
        ForecastResponse response = new ForecastResponse()
            .forecast(new Forecast(location, temperature));
        eventSink.send(
            sseDataEventBuilder
                    .data(response)
                    .id(String.valueOf(eventId))
                    .build());
    };
    eventSink.send(
        sse.newEventBuilder()
            .id("completed")   // final event
            .mediaType(MediaType.APPLICATION_JSON_TYPE)
            .data("{}")        // empty JSON data required by some browsers
            .build());
    eventSink.close();
}
```

This code even does a little bit more in order to support reconnection to an SSE event stream in case of a network failure. This allows clients to resume receiving events from the point when the connection was interrupted, and is supported by the SSE protocol out of the box. We'll talk more about automatic reconnection with SSE later.

Invoking the SSE service

Accessing our new SSE forecast service is very simple. The SSE protocol uses the same underlying HTTP protocol as any other REST service; therefore, it can be invoked just like any other REST service. The only difference is that the response would be in a different format than JSON. You can quickly observe it when you invoke the service from a REST client, such as Postman, with a `GET` method at `http://localhost:8080/forecast-service-async/smartcity/forecast`.

Here is the result of invoking the Forecast service in Postman:



http://localhost:8080/forecast-service-async/smartcity/forecast

```
id: 1
data: {"forecast":{"location":{"name":"London"},"temperature":{"temperature":34.0,"temperatureScale":"CELSIUS"}},"processingTime":517}

id: 2
data: {"forecast":{"location":{"name":"Istanbul"},"temperature":{"temperature":33.0,"temperatureScale":"CELSIUS"}},"processingTime":1025}

id: 3
data: {"forecast":{"location":{"name":"Prague"},"temperature":{"temperature":31.0,"temperatureScale":"CELSIUS"}},"processingTime":1532}

id: completed
data: {}
```

You can see that the response contains multiple chunks that contain data in JSON format. These represent all the SSE events sent by the service. Postman doesn't recognize them as events and waits until the response is complete.

On the other hand, if you invoke the service using the `curl` command-line utility, which recognizes SSE events in the response, you will see a different behavior.

When you invoke the `curl` utility from the command line, you'll see the following output:

```
> curl http://localhost:8080/forecast-service-async/smartcity/forecast
id: 1
data:
{"forecast":{"location":{"name":"London"},"temperature":{"temperature":37.0
,"temperatureScale":"CELSIUS"}},"processingTime":515}

id: 2
data:
{"forecast":{"location":{"name":"Istanbul"},"temperature":{"temperature":44
.0,"temperatureScale":"CELSIUS"}},"processingTime":1028}

id: 3
data:
{"forecast":{"location":{"name":"Prague"},"temperature":{"temperature":44.0
,"temperatureScale":"CELSIUS"}},"processingTime":1538}

id: completed
data: {}
```

The response will be in the same format, but the chunks for separate events will be printed immediately when they arrive, without waiting until the response is completed. This is because the `cURL` utility recognizes the `"text/event-stream"` media type of the response and understands the format of the response.

Invoking the SSE service from JavaScript

It's very common to invoke SSE services from JavaScript code in a browser to update the page as the keep coming from the service. Most modern browsers support the SSE protocol and provide the `EventSource` object, which makes invoking an SSE service very simple.

To invoke our forecast service, we only need the following code:

```
var source = new
EventSource("http://localhost:8080/forecast-service-async/" +
   "smartcity/forecast");
source.onmessage = function (event) {
   if (event.lastEventId === "completed") {
      source.close();
   } else {
      alert("Received: " + event.data);
   }
};
```

The preceding code will do the following:

- Create an instance of the `EventSource` object and immediately open a connection to the SSE service.
- Provide a function to handle messages as the `onmessage` property of our `EventSource` instance.
- The handler validates the value of `lastEventId` to detect the final message, and closes the `EventSource` object when the final message is received.

TIP

If a browser doesn't provide the EventSource object with access to SSE services, it's possible to use a polyfill library, which will seamlessly add it. A collection of various polyfill libraries can be found in a wiki of the Modernizr project at `https://github.com/Modernizr/Modernizr/wiki`.

Building an SSE Java client

To consume our asynchronous SSE forecast service from Java code, we're going to create a standalone Java application that will connect to it and print the data received as events. We're again going to use the JAX-RS client API we used in the previous chapter, `Chapter 3`, *Connecting Microservices Together*, to access a standard REST service.

Remember that SSE is just an extension of the standard HTTP protocol, and the JAX-RS client API for SSE also works as an extension to the standard client API. We're going to use the `SseEventSource` class from the JAX-RS API to extend the standard `WebTarget` object and to add listeners to handle SSE events.

To build our SSE client, we need to create a Java project and add some required dependencies that provide the Java EE functionality to a plain Java application. We'll need the following dependencies:

- `jersey-client`
- `jersey-hk2`
- `jersey-media-json-binding`
- `jersey-media-sse`

The first three are the same dependencies that were needed in the *Building a Client* section of the `Chapter 3`, *Connecting Microservices Together*. The only new dependency is `jersey-media-sse`, which provides the SSE portion of the standard JAX-RS client API.

If you want to invoke an SSE service from a web application deployed to a Java EE container, all of the required dependencies are provided by the container, and none of the Jersey dependencies are needed in your application.

The Maven dependency for `jersey-media-sse` can be added as follows:

```
<dependency>
   <groupId>org.glassfish.jersey.media</groupId>
   <artifactId>jersey-media-sse</artifactId>
   <version>2.26</version>
</dependency>
```

In the Java client, we'll create an instance of `SseEventSource`. We'll do this by first creating an SSE builder from an instance of `webTarget` using the static `target` method. This builder will then create an instance of `SseEventSource` while it allows us to configure the event source before it's created.

This is the simplest example of creating `SseEventSource` from an instance of `webTarget`, which is stored in the `webTarget` variable:

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

Unlike the JavaScript `EventSource` object, the creation of an `SseEventSource` instance won't open the connection automatically. We should first call the `register` method to register an event callback and, optionally, error and completion callbacks. Only then will we open the connection with the `open` method.

Use the `register` and `open` methods to initiate a connection, and register callbacks, as follows:

```
sseEventSource.register(onEvent, onError, onComplete);
sseEventSource.open();
```

The callbacks passed to the `register` method can be simple lambda expressions that implement particular functional interfaces, as follows:

- `Consumer<InboundSseEvent> onEvent`
- `Consumer<Throwable> onError`
- `Runnable onComplete`

Finally, we need to make sure that both the `SseEventSource` instance and the original JAX-RS `Client` instance are closed once the client stops consuming SSE events in order to close the connection and release unused resources. In order to synchronize the main thread of our Java client with the callbacks that are going to be executed in different threads, we'll use `CompletableFuture`, which provides a simple thread-safe API to complete execution and wait for its completion in a different thread. If `CompletableFuture.join()` is called, the main thread will wait until SSE processing is finished.

The client can receive the following events:

- The `onComplete` callback is triggered, meaning the client detected the end of the response.
- The `onError` callback is triggered, meaning the client detected an unrecoverable error.
- The `onEvent` callback is triggered by a termination event, meaning that the server sent the last event.

The last condition doesn't apply to every SSE service. The Forecast service is designed to send a termination event with a special event ID so that the client can easily detect the end of the event stream. This is because we've seen that SSE clients detect the end of SSE event stream ambiguously, sometimes hanging and waiting for more events to come.

Here is the complete code for `ForecastResourceSseClient.java`, which is a standalone Java program that connects to the Forecast service and prints the received data, as follows:

```
public class ForecastResourceSseClient {
   public static void main(String... args) throws Exception {
      Client restClient = ClientBuilder.newClient();
      WebTarget target = restClient.target("http://localhost:8080/"
         + "forecast-service-async/smartcity/forecast");
      SseEventSource sseEventSource = SseEventSource.target(target)
         .build();
      CompletableFuture<String> asyncProcessing = new
CompletableFuture<>();

      Consumer<InboundSseEvent> onEvent = (InboundSseEvent event) -> {
         if ("completed".equals(event.getId())) {
            asyncProcessing.complete("complete event received");
         } else {
            ForecastResponse forecastResponse =
event.readData(ForecastResponse.class,
               MediaType.APPLICATION_JSON_TYPE);
            System.out.println("Event received: " + forecastResponse);
         }
      };
      Consumer<Throwable> onError = (Throwable error) -> {
         asyncProcessing.completeExceptionally(error);
      };
      Runnable onComplete = () -> {
         asyncProcessing.complete("OnComplete");
      };
      sseEventSource.register(onEvent, onError, onComplete);
      sseEventSource.open();
      asyncProcessing.join();
      sseEventSource.close();
      restClient.close();
   }
}
```

The SSE client can be summarized into the following steps:

1. A standard JAX-RS `WebTarget` is created for the URL of an SSE service.
2. An `SseEventSource` is created from `WebTarget` using `SseEventSource.target`.
3. An instance of `CompletableFuture` is created for later synchronization.
4. Callbacks are attached using the `register` method; callbacks complete the future appropriately.
5. The SSE service is invoked by calling the `open` method.
6. The main thread waits until the SSE event stream is processed by waiting for the future to complete.
7. The `SseEventSource` and `Client` resources are closed.

When using `CompletableFuture`, as in our example, we can chain closing of the `SseEventSource` and `Client` resources right after the future call is completed, with a method such as `thenRun` of `CompletableFuture`. This will ensure that the resources are closed without the main thread waiting for the completion of the future.

Alternatively, we can chain the closing of resources after the future is completed with the `thenRun` method, like this:

```
asyncProcessing.thenRun(() -> {
   sseEventSource.close();
   restClient.close();
})
```

Automatically reconnecting of SSE clients

One of the standard features of SSE, which works out of the box, is automatic reconnection. SSE clients always try reconnecting to the same stream of events when a connection is unexpectedly lost. If the server includes the ID attribute in events, then the client will automatically include the last known ID into the new connection. The client adds it to a new request as a value of the `LAST_EVENT_ID` HTTP header. This may be used to start sending new events to the client from this ID, or replay all past events that happened since an event with this ID was sent.

Special care has to be taken if we can't handle reconnection on the server. Ignoring the `LAST_EVENT_ID` header may lead to multiple invocations of our service in case of a failure. In case the service is supposed to send a sequence of requested results, such as our Forecast service, a connection failure and subsequent reconnection would cause all the results to be sent again. If the header is present in the request, we should take it into account and only send the events that haven't been received by the client. If that's not possible – for example, if the order of events isn't always the same, we should ignore all requests that contain the header because they come from a reconnect. This would disable the reconnect feature and a call to the service would fail and disconnect immediately after a failure message is received from the server without any chance to recover automatically.

The synchronous version of the Forecast service can easily support auto-reconnect. It executes external services in the same order, one after the other, and the order of results is always the same. Therefore, we can skip all the results that were already sent.

Any asynchronous version of the Forecast service that calls external services in parallel produces events in an unpredictable and non-deterministic order. It's then very hard to find out which events were sent based just on the ID of the last request. To avoid calling the same service repeatedly upon connection failure, we ignore all reconnection requests by requiring that the `LAST_EVENT_ID` header is empty, as follows:

```
public void getLocationsWithTemperature(
    @Null @HeaderParam(HttpHeaders.LAST_EVENT_ID_HEADER) Integer
lastEventId, …)
```

Even if we won't use the value of `LAST_EVENT_ID` in the method, declaring it as a header with `@HeaderParam` and placing the `@Null` validation constraint allows us to execute the method only if the header is empty. If a connection is dropped during receiving events, clients that attempt to reconnect would receive an error status and would stop reconnecting further.

In order to support reconnecting asynchronous calls correctly, the event ID would have to encode all events already sent, to avoid sending them again. However, this can get too complicated and the event ID would grow with the number of events sent. Therefore, it's simpler and safer to raise an error and make the client handle the problem, for example, by restarting the connection from the beginning.

SSE is often used to send an infinite stream of events to which clients can connect at any time. This is where auto-reconnect makes the most sense and helps a lot. Receivers would just reconnect and continue receiving future events as they are produced. This pattern would fit a scenario where the Temperature service should send the current temperature at a regular interval.

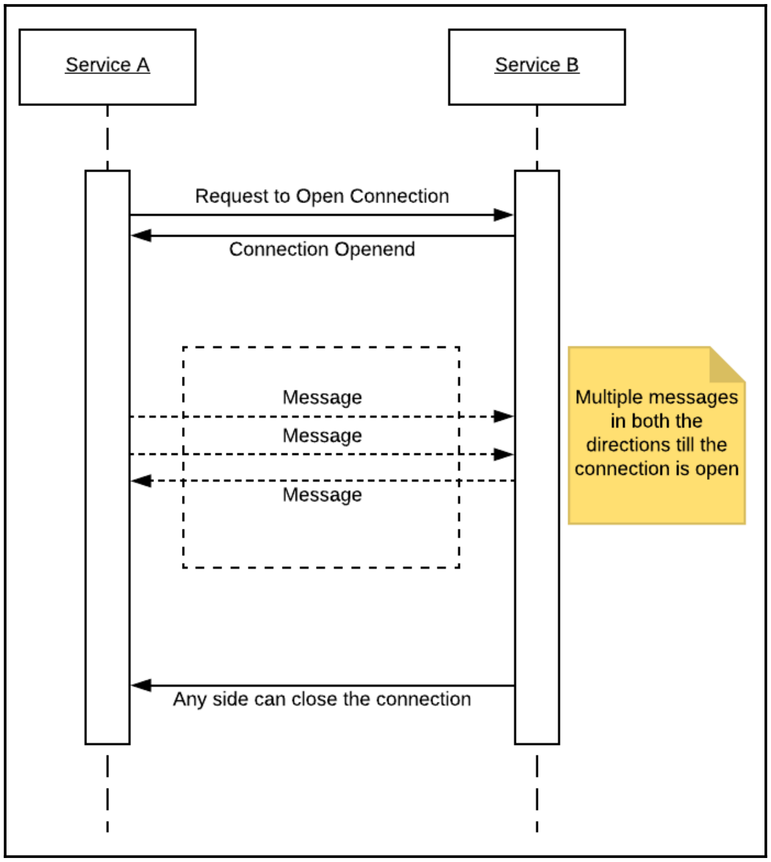
For building SSE endpoints that produce an infinite stream of events, JAX-RS provides `SseBroadcaster`, which you can use to distribute events from a single source of data to every connected SSE client.

Two-way asynchronous services with WebSocket

So far, we've improved our Forecast service asynchronously, to communicate with other services and to send data to the client. What we haven't changed is the underlying HTTP protocol, which is used to invoke REST services and SSE endpoints. If we want to optimize the communication further, we have to start thinking about the limitations of HTTP and replacing it with something more powerful and efficient.

Another technology that can help us improve our performance is WebSocket, which is also a web standard, like HTTP. Unlike HTTP, it supports two-way (full-duplex) communication over a single session. This allows us to initiate multiple subsequent calls to a web service, without creating a separate session for each, and it also allows us to control the flow of data to or from the service in a single session. WebSocket also allows sending streams of data, such as SSE, and it allows us to send both text and binary data. This is unlike SSE, which requires the data to be sent in text form.

The following diagram shows how WebSocket-based communication works:


Service A
Service B
Request to Open Connection
Connection Openend
Message
Message
Message
Multiple messages in both the directions till the connection is open
Any side can close the connection

A quick comparison between HTTP and WebSockets

There are several reasons why the HTTP protocol is so widespread: it's the protocol used by the web; there are many tools that support working with it; and it's textual and very straightforward, which makes it easy to understand and troubleshoot even without complex tools. In Microservices architectures, it's often the primary choice for communication because it's also easy to monitor, secure, and control in distributed environments. However, it has multiple limitations that make it less efficient than other alternatives.

The most obvious downside of the HTTP protocol is that it allows only one-way communication. Although HTTP/2 allows some two-way communication, it's limited only for the purpose of requesting multiple resources more efficiently.

Another drawback is that the protocol is inherently textual. Although it's possible to send binary data over plain HTTP, the data is always wrapped in a text envelope, which means that a lot of metadata about the request and response are still in text form. Furthermore, SSE only allows us to send text data.

On the other hand, the advantage of the HTTP protocol is its simplicity and transparency. Because it's so widespread, it works well with firewalls and often works out of the box in any infrastructure. WebSockets, which don't have the limitations of the HTTP protocol, are often limited by the infrastructure. They require a separate port to create connections and often require adjustments in the infrastructure to allow us to establish a connection. Unlike HTTP, WebSocket is designed to keep long-lived TCP connections. The WebSocket-based communication consumes more resources and causes some trouble if the connections are dropped by the infrastructure policy or a failure. This is not a big issue with SSE, which expects connection failures and includes an inherent auto-reconnect mechanism.

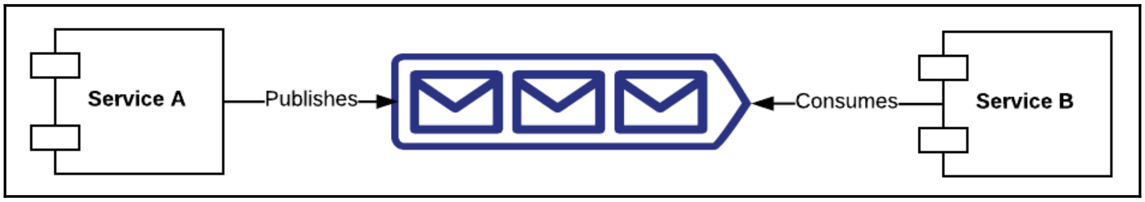
In the end, WebSocket provides a lot more flexibility than REST or SSE endpoints

Decoupling services with message-oriented middleware

So far in this chapter, we've discussed direct communication between client and server or between services. Another important way to implement asynchronous communication is through the use of middleware, or more specifically, by implementing asynchronous communication through the use of queues. To start with, let's take a classic publisher-consumer problem, where service A is responsible for generating some information, which is to be consumed by another service, B. Though the communication can be synchronous as well, where we can configure our application in a way that the communication is deemed completed when a successful acknowledgment is received by calling the service, this kind of communication is best suited to asynchronous communication.

We can implement two major types of communications patterns using messaging: point-to-point and publisher-subscriber. In point-to-point communication, as the name suggests, we have two endpoints. One would serve as publisher, sending messages to the listener point, known as a consumer.

The following diagram shows a simple point-to-point communication:



A simple example is an order-management service and an inventory-management service in an e-commerce system. Whenever an order is executed, the order-management service would add a message to the queue that the order was successfully executed. The message is read by the inventory service, which in turns updates inventory. The advantage of using a queue-based approach is that it removes any dependency between the two services.

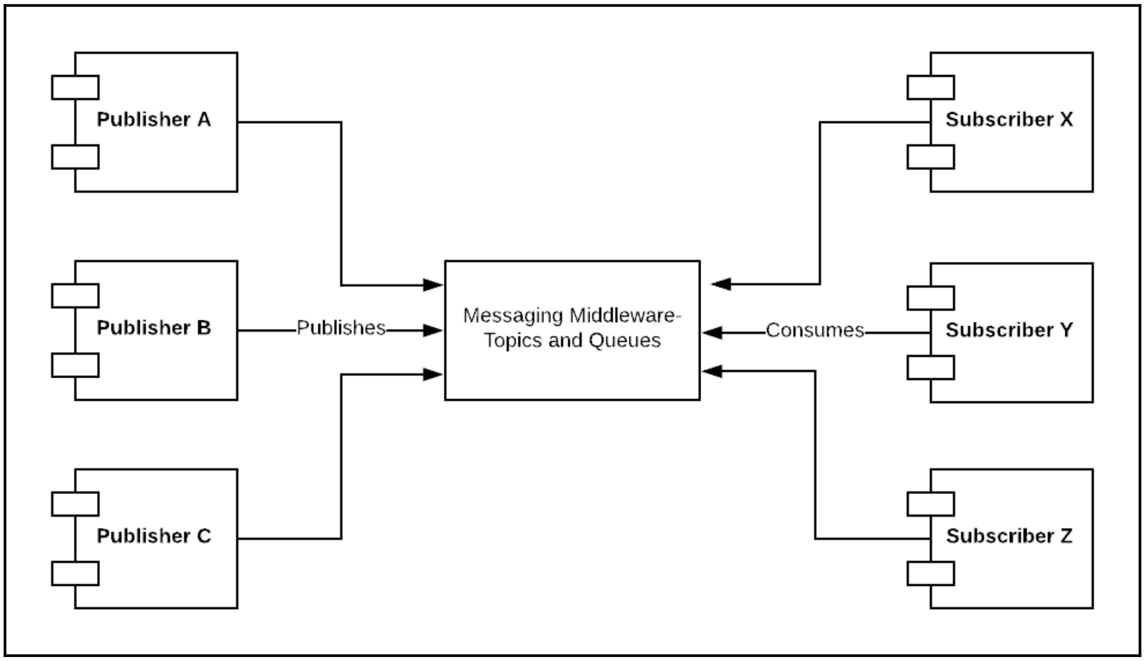
The order-management service doesn't need be aware of the inventory service. All it is aware of is a middleware queue that it needs to add the message to; what happens with the message is not its responsibility. So, even if an inventory management service is down or broken, the order management will work fine without any disruption. The messages will be available in the queue, and when the inventory service is fixed, it will consume all the messages.

This example highlights a couple of important aspects of this kind of communication. We can see very clearly that order-management system will keep on working without even knowing whether inventory is being updated properly. It is important to understand our use case: do we actually want this level of decoupling? This brings another aspect, which is that even if the inventory service were down for some time, we have actually not missed any updates. When the service was up, it consumed all the messages and data was eventually corrected.

If we were just logging the order details for reporting purposes, we could have used queue-based communication without giving it a second thought. But, if our use case can result in incorrect data in the system, we should be carefully accessing our tolerance of the delay in receiving the data.

So far, we've looked into cases where we had one publisher and one consumer. Say we have a case where more than one services wants to take an action on a message. In the preceding example, we have both log-management and inventory-management systems, which are interested in an order-completion event. In such cases, we use an approach known as publisher-subscriber or pub-sub. In this approach, instead of publishing to a queue, we publish to a specific topic – for example, order-completion can be a topic. Multiple publishers can publish to a topic and multiple subscribers can listen to a topic.

The following diagram shows a pub-sub architecture:



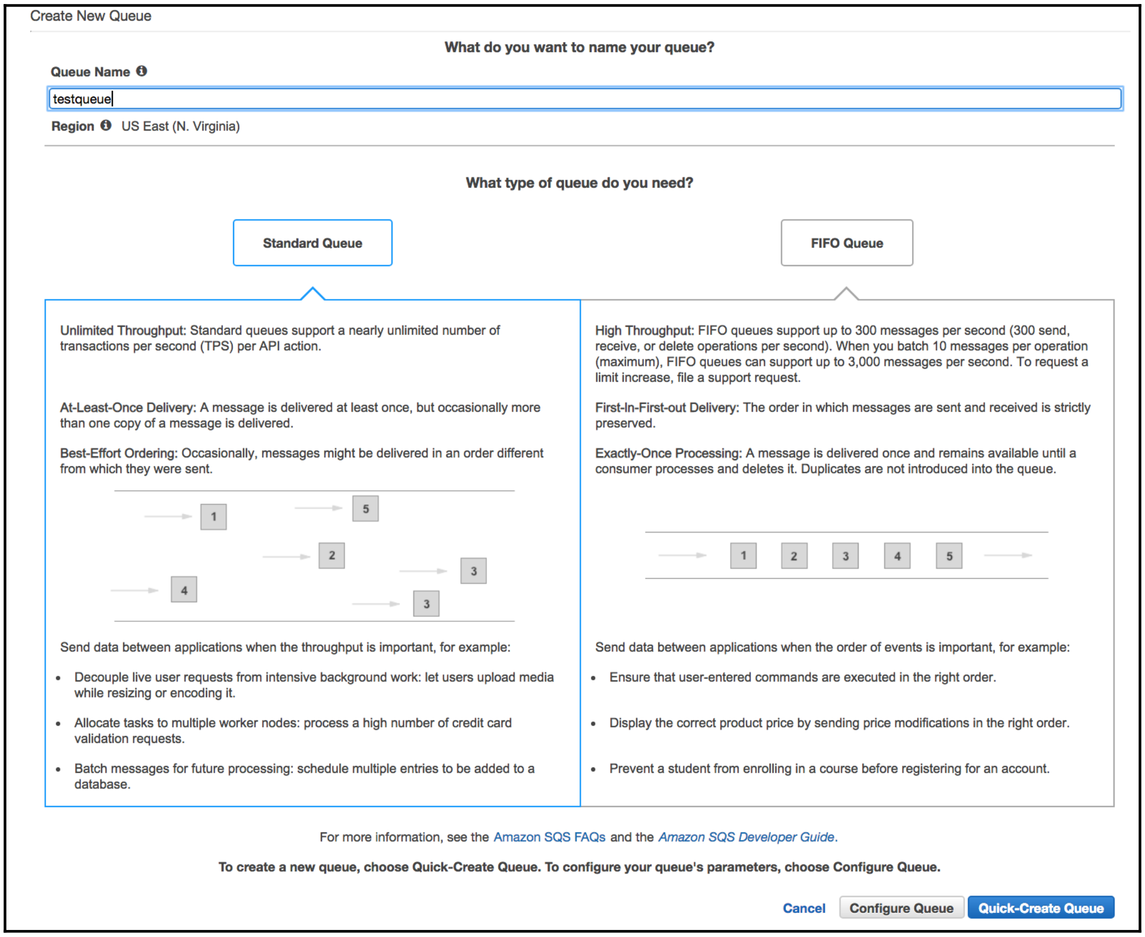
In our example case, the order-management service would be publishing a message to the order completed, whenever an order is executed successfully, and services, such as logging service and inventory-management service, listen on this topic. Whenever a new message is published, these services consume the message and take appropriate action.

An example of message-oriented middleware

Let's look at a code example that publishes and consumes a message. **Java Message Service** (**JMS**) has been the industry standard for years. For sake of this example, we will use Amazon's **Simple Queue Service** (**SQS**) to avoid the additional overhead of setting up middleware and the queue infrastructure.

AWS provides us with a simple mechanism to create the queue, but you can use any provider of your choice.

The following screenshot shows how a queue is created in AWS-SQS:



Create New Queue

What do you want to name your queue?

Queue Name testqueue

Region US East (N. Virginia)

What type of queue do you need?

Unlimited Throughput: Standard queues support a nearly unlimited number of transactions per second (TPS) per API action.

At-Least-Once Delivery: A message is delivered at least once, but occasionally more than one copy of a message is delivered.

Best-Effort Ordering: Occasionally, messages might be delivered in an order different from which they were sent.

Send data between applications when the throughput is important, for example:

- Decouple live user requests from intensive background work: let users upload media while resizing or encoding it.
- Allocate tasks to multiple worker nodes: process a high number of credit card validation requests.
- Batch messages for future processing: schedule multiple entries to be added to a database.

High Throughput: FIFO queues support up to 300 messages per second (300 send, receive, or delete operations per second). When you batch 10 messages per operation (maximum), FIFO queues can support up to 3,000 messages per second. To request a limit increase, file a support request.

First-In-First-out Delivery: The order in which messages are sent and received is strictly preserved.

Exactly-Once Processing: A message is delivered once and remains available until a consumer processes and deletes it. Duplicates are not introduced into the queue.

Send data between applications when the order of events is important, for example:

- Ensure that user-entered commands are executed in the right order.
- Display the correct product price by sending price modifications in the right order.
- Prevent a student from enrolling in a course before registering for an account.

For more information, see the Amazon SQS FAQs and the *Amazon SQS Developer Guide*.

To create a new queue, choose Quick-Create Queue. To configure your queue's parameters, choose Configure Queue.

As you can see, all we need to provide is a queue name and a Quick Create Queue. If required, you can update default configurations, such as how long you want the message to be available in the queue, by clicking the **Configure Queue** button. Also, you can use the FIFO queue, which guarantees the order of messages being sent is in the form of "first in first out." For the sake of this example, we will use the default configurations.

Here is the code to demonstrate the read and write operations on the AWS queue:

```
import java.util.List;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.AWSCredentialsProvider;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.internal.StaticCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.Message;
import com.amazonaws.services.sqs.model.SendMessageRequest;
public class SendReceiveMessages
{
  public static void main(String[] args)
  {
    AWSCredentialsProvider provider;
    AWSCredentials credentials = newBasicAWSCredentials("Key","Secret");
    provider = new StaticCredentialsProvider(credentials);
    AmazonSQS sqs =
AmazonSQSClientBuilder.standard().withCredentials(provider).withRegion(Regi
ons.US_EAST_1).build();
    String queueUrl =
"https://sqs.us-east-1.amazonaws.com/305881070752/TestQueue";
    SendMessageRequest send_msg_request = new SendMessageRequest()
.withQueueUrl("https://sqs.us-east-1.amazonaws.com/305881070752/TestQueue")
.withMessageBody("hello world")
.withDelaySeconds(5);
    sqs.sendMessage(send_msg_request);
    // receive messages from the queue
    List<Message> messages = sqs.receiveMessage(queueUrl).getMessages();
    System.out.println("message:"+messages.toString());
    // delete messages from the queue
    for (Message m : messages)
    {
      sqs.deleteMessage(queueUrl, m.getReceiptHandle());
    }
   }
 }
```

This code sends and receives a message from the AWS queue. This is a straightforward implementation of point-to-point communication. AWS also supports the creation of topics through its **Simple Notification Service** (**SNS**). These topics can then be attached to queues, so when a new message comes into the system for a topic, it's delivered to all the queues listening to that topic, and hence delivered to listeners of the queue.

Summary

In this chapter, we discussed the importance of asynchronous communication when using a Microservices-based design. We discussed various ways of implementing asynchronous communication. We started by discussing a simple REST-based implementation of APS, which helps us to communicate asynchronously among services. Then we looked at web sockets for maintaining persistent connections. This kind of communication is good when we need to send multiple messages asynchronously among services. Finally, we looked at message-oriented middleware to implement asynchronous communication, and hence decouple our services. This kind of communication helps us send messages to a single service through point-to-point communication or multiple services through the publisher-subscriber pattern.

In the next chapter, we will discuss best practices for building a robust set of MicroServices.

5
Path to Robust Microservices

So far, we've talked about Microservices and how they can help us create stable applications. But, we need to understand that Microservices do bring their own challenges. Since we now are dealing with multiple services that are executing independently of each other, rather than a single application, we need to make sure that there is no single point of failure. We need to make sure that a problem in one service does not impact the system as a whole. Additionally, how should we handle failures in our application? Can our application self-heal? In this chapter, we will talk about some of the techniques that will help us to create a robust application using Microservices.

We will cover the following topics in this chapter:

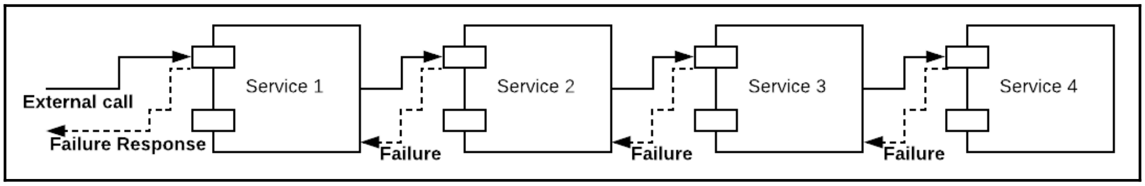
- Building for failure
- Isolating the failure
- Handling the failure
- Recovering from failure
- Preparing for failure

Building for failure

No software can be 100% error-free, but we always try to make it as stable as possible while ensuring that if an error occurs, it will be handled gracefully. You must have heard the term build for failure. The idea is that you cannot assume that your applications or services will never fail. Instead, you should assume that they will fail no matter how carefully you have built and tested your services.

There are different types of failures. A common failure case is when the developer has missed handling some edge condition such as an invalid value received as input, or a memory leak due to too many unused objects, or your application is facing more load than expected. Then we can have hardware failures, where a server or a cloud node goes down. We cannot handle all issues beforehand, so we need to plan for failure cases and design the architecture so that our overall service is stable even if an issue occurs.

A cascading failure would look as follows:



There are techniques such as an implementation of the circuit breaker pattern,the fail fast pattern, the fan out pattern, and so on, which can help us manage this kind of failure.We will discuss these later in this chapter in the *Handling the failure* section.

So far, we've discussed why it is important to know that there are various best practices and patterns that we can (and should) follow in order to make our services robust and ready for failure. We will discuss the important practices in the rest of this chapter.

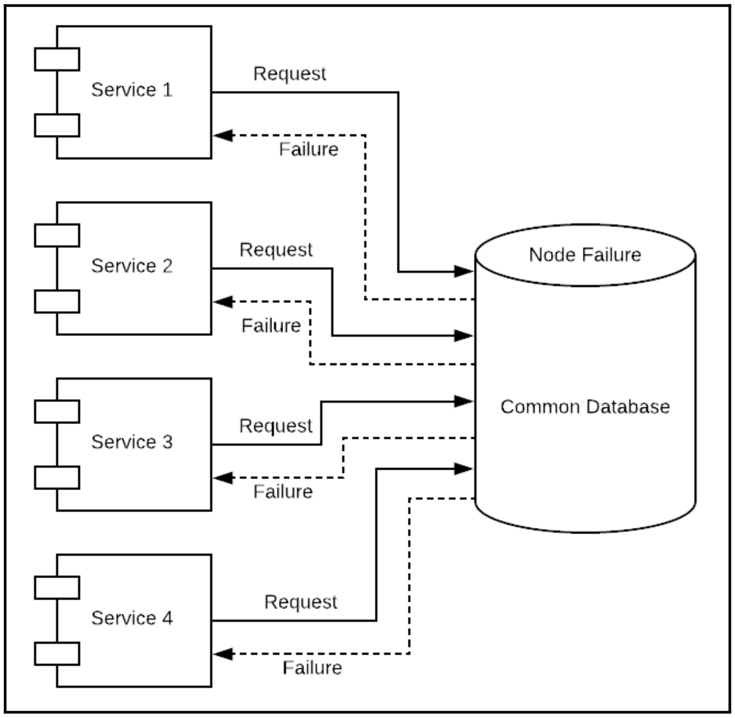
Isolating the failure

If you can take only one thing from this chapter, take this always design your system in a way that failure in one service or business area should not get propagated to other areas. In short, isolate the failure.

Years ago, plugging in a faulty electronic device at home could cause a house-wide power outage. It could even cause a short circuit or fire. Then came **Miniature Circuit Breakers** (**MCB**), so if something goes wrong, only a single MCB that pertains to the area where the faulty device was used would trip. This is a good example of isolating the failure; if something goes wrong in one area, it is not able to impact other areas. Not surprisingly, we have a pattern for failure handling in services, called the **circuit breaker pattern**, which we will discuss later in this chapter in the *Handling the failure* section.

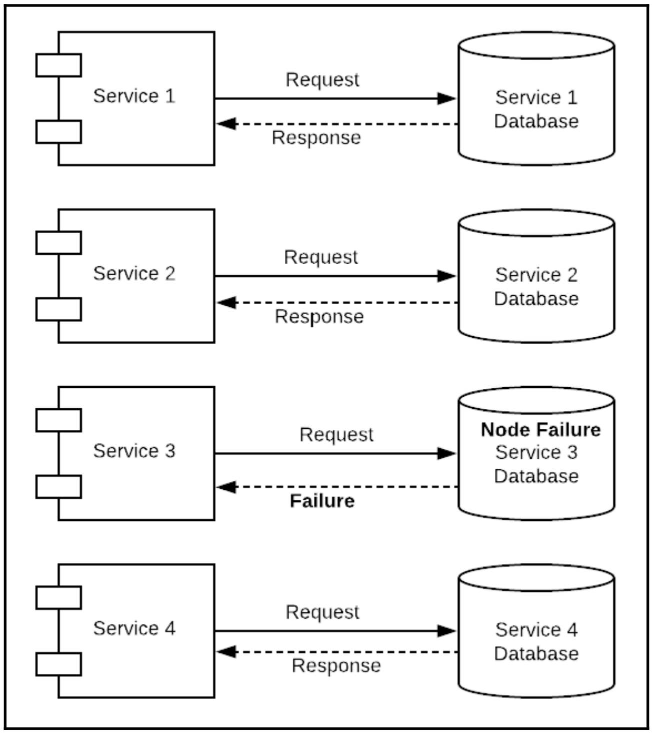
What I am trying to emphasize here is that one should make sure that there is no single point of failure in the system, or that a failure in one service should not impact other services. One simple example of having a single point of failure is depending on a single database server. If the database server is down, the whole system is down. The idea behind Microservices is that each service should be looked at as an independent unit.

The following scenario depicts a case where, due to a failed common database node, all the services dependent on it have failed too:



Clearly, we have not implemented the Microservices architecture properly and are not taking advantage of using Microservices. Microservices are supposed to be built independently, in a manner that failure in one does not cause failure in the others.

Let's revisit the problem with an updated design:



We can see that, if properly isolated, a failure will only impact a single service. In the preceding diagram, we have isolated all four services, and a failure in the database for service three only impacts service three; the other services keep on working smoothly.

Failure isolation needs to be handled at multiple levels; we need to handle it at the service level itself. So if there is a buggy service, say a search service is buggy in an e-commerce site, it should not impact the checkout or catalog view. Additionally, we need to handle failure at the service instance level. For example, if we had five instances of a search service, and one of them goes down, it should not impact the other four instances.

When starting to build an application, you need to design for isolation. We need to make sure that even if a problem occurs, it is isolated and does not impact the system as a whole. In short, we need to avoid cascading failures. The age-old principle of Low Coupling and High Cohesion helps here. That is, only the features that are alike and must be put together should be together; otherwise, we should keep them separate.

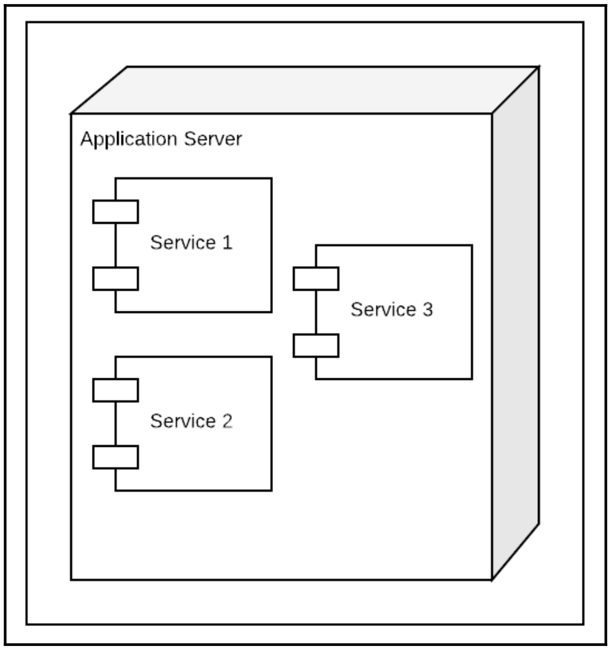
We will now talk about concrete patterns and techniques for how to isolate a failure.

The bulkhead pattern

The bulkhead is the most common and basic pattern when we discuss isolation. The term itself has an interesting origin. The term is borrowed from cargo ships. A bulkhead is a wall built between different cargo sections in the ship. This wall ensures that if one of the sections gets flooded or catches fire, other sections do not get impacted. The idea is to build a similar partition between our services, so that failure in one area does not propagate to others.

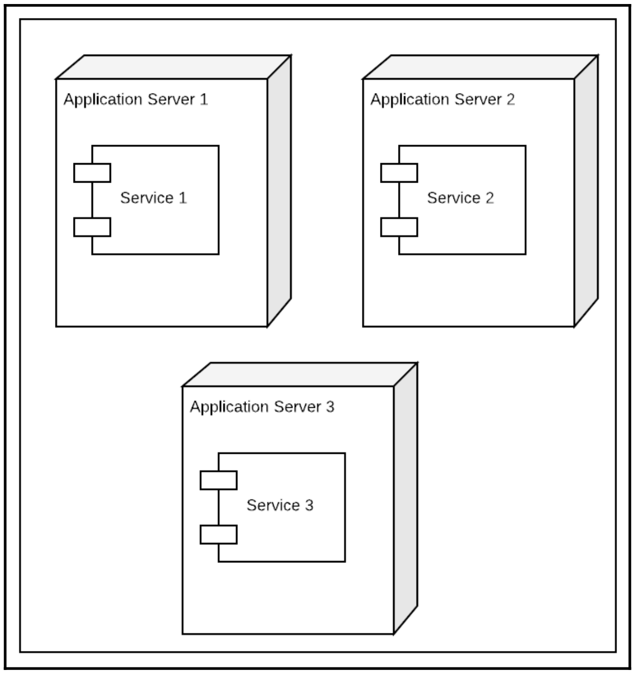
We need to implement a bulkhead pattern at multiple levels, development as well as deployment. While developing, we need to make sure that we build our services in a manner that they are logically independent of each other. Moreover, while deploying, we need to make sure that we deploy each service independently so one node failure does not bring down all critical services.

Let's look at the following simple example to understand the power of the bulkhead pattern:



Say we have deployed multiple services in the same application server. Now, an application server can manage a fixed number of connections at a time. Say one of the services has buggy code, slowing it down, which results in blocking the threads for a longer duration. This will cause other services to suffer as they will not have adequate resources.

To tackle this, we need to use the bulkhead approach and create isolated blocks:



We have handled the scenario by isolating each service. Now, if one of the services gets into trouble and blocks all the threads, we will end up losing only one service and the other services will keep working fine.

Stateless services

It is important to understand the power of stateless services. Think of a scenario where a service is maintaining some kind of state; say a user can fill in form 2 only if they have filled in form 1, and we are trying to keep this information within the application, say in a session. The problem is that we will need to make sure that the same service instance receives a request from the same user. There are solutions, such as sticky sessions or replication of session, but they add to the overhead and complexity of the system.

Using stateful services makes it difficult to isolate the failure. If the service is in an error state or responding slowly, it is not easy to redirect the traffic to some other node or simply restart the node, as we will lose the information state information maintained locally.

We will talk more about stateless services with an example in `Chapter 6`, *Scaling Microservices,* when we discuss scaling Microservices.

The robustness principle

"Be liberal in what you accept, and conservative in what you send".

Source for this statement: `https://devopedia.org/postel-s-law`

Though at first glance this does not seem to be directly related to the topic of isolation, this age-old principle helps to define the core of Microservices.

Let's break the statement down into two parts:

"Be conservative in what you send".

This is what Microservices are all about: focusing on a fixed set of responsibilities and being responsible for communicating with other services. When we say to be conservative in what you send, we are also accepting the fact that we are doing a limited, fixed set of activities in the service. Ideally, one Microservice should be focusing on one task. So, when we say we have a TaxCalculator service or a PhotoUploader service, we know that the service is going to calculate tax or upload a photo, and nothing else. In other words, this is in sync with our Single Responsibility Principle, that is, one service is handling one responsibility. This is a very important concept when it comes to implementing isolation. Let's say we have a service, EmployeeDataUpdateAndPhotoUploader. For some reason, the photo-uploader part is broken (say due to a developer mistake, it goes into a loop or out of memory). Now the photo-uploader part can slow down or break the employee-data-update part as well. So if all I needed was to update the address for an employee, I cannot do it because the photo-uploader code is broken.

Whenever you feel that your service is doing too much work, break it into multiple services. A simple rule is to try to write down the role that the service is playing. If there are too many ands, for example, we are writing that this service does this and this, you know it will be difficult to manage.

Let's look at the other part of the rule.

"Be liberal in what you accept from others."

You cannot control how your service is going to be used, but you can manage the behavior on your end, by keeping a check on which input your service is going to accept. So if your service needs four parameters to work properly, build it in a manner that it can receive three or five parameters. What should our service do in case incorrect input is recieved? Can we somehow manage and convert the input into the required format? If not, we should gracefully send back the required messages. The idea is that your service should never be caught unprepared. Unplanned input can break down your service to get infinite loops or a memory leak situation, where one faulty request is making others suffer.

Handling the failure

In the *Isolating the failure* section, we talked about practices that help us to isolate the failure. That is, if service is a failing, it should fail independently without impacting the rest of the system. But, just isolation might not be a complete solution in itself. We still need to handle the failure. We need to see how our application will behave if a service fails.

There are multiple ways in which we can make sure we handle failures gracefully. We will discuss some of these options now, starting with Asynchronous Communication.

Asynchronous communication

Whenever possible, we should go for asynchronous communication with clients and other services. Asynchronous communication helps us to decouple code and does not block the calling service. For example, if your frontend UI goes for asynchronous communication with backend services, we make sure the system remains usable while we are loading the required data at the backend. The end user is still able to interact with the system. Even if a service or two are failing, we are making sure other services are returning information and are usable by the end user even if the unresponsive service blocks a couple of threads.

We should use an asynchronous queue or middleware-based communication when an immediate action or response is not required. Say we are logging; we will add the data to queue or broker instead of the service directly.

Fail fast

This is an important concept in today's world, where performance is of the utmost importance. A very simple example would be that you hit a link on a website, it starts loading the page, shows you a loader on a blank page, you wait patiently for seconds or maybe for minutes, and then you get the message Could not retrieve data, please try again later. It's like rubbing salt in a wound. If you had to show the error message, why did you make the user wait?

So if the service has to fail, it should fail fast. But how do we achieve that? There are multiple ways, based on your service. If your service is going to call another service, you can keep a check on the health of the other service, maybe by a handshake or ping call, so you are aware of the fact that the other service is in a position to handle your calls.

An additional approach is the use of the bounded queue. Any request reaching to the service is queued, and this queue is bounded. So, if we know that our service can handle 10,000 calls in an acceptable amount of time without getting slow, we will only queue 10,000 requests and then reject additional requests. This way, the additional calling services can retry after some time and do not get blocked.

Timeouts

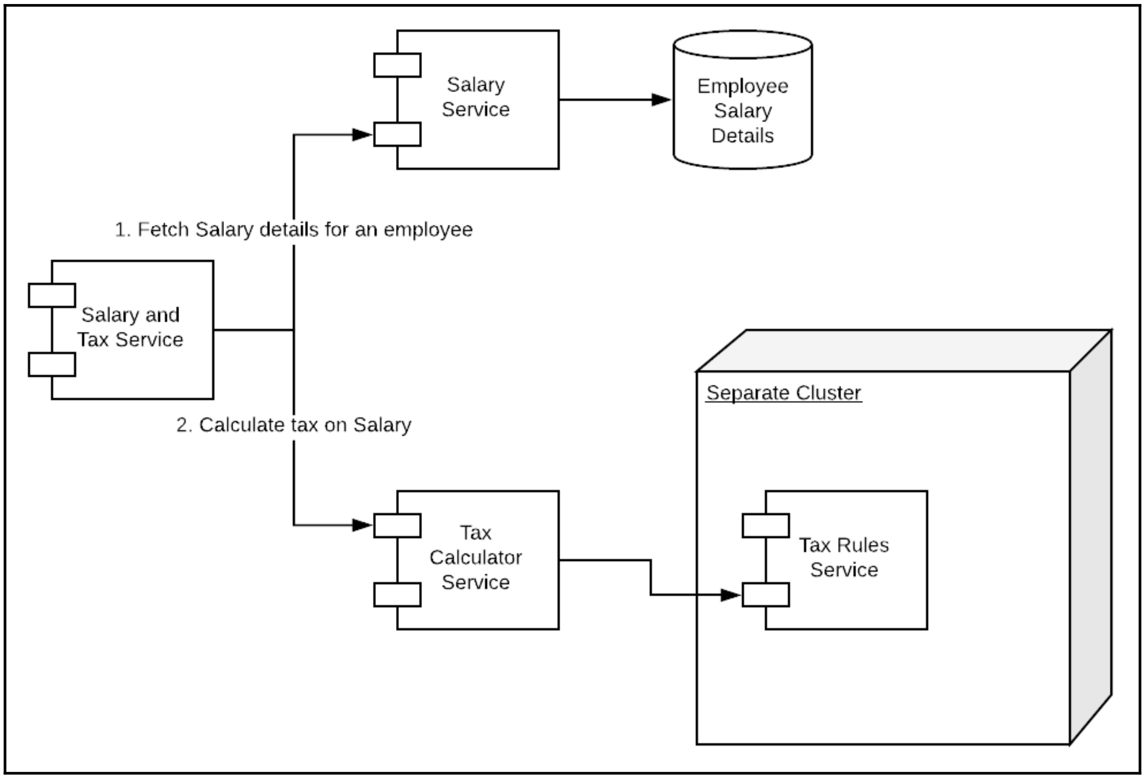
Timeouts are a simple yet very powerful tool for handling failures in calling services that are into an error condition. For example, if we are calling a service, that is facing a performance issue, our timeout checks at the calling service end will save us from the pain of waiting infinitely for a response. This is in sync with the fail fast approach, but this time we are implementing the solution at the client side or at the calling service end.

Circuit breakers

A circuit breaker is a term borrowed from electronic circuits. Whenever a circuit gets in a faulty situation, the role of a circuit breaker is to cut it off from the supply to make sure it does not impact other areas. In other words, the problem is restricted to one part and cannot propagate. Using the same idea, we are talking about a technique to develop over Microservices in such a way that a problem in one Microservice should not propagate to other services, and that the impact of a failure is restricted to one area.

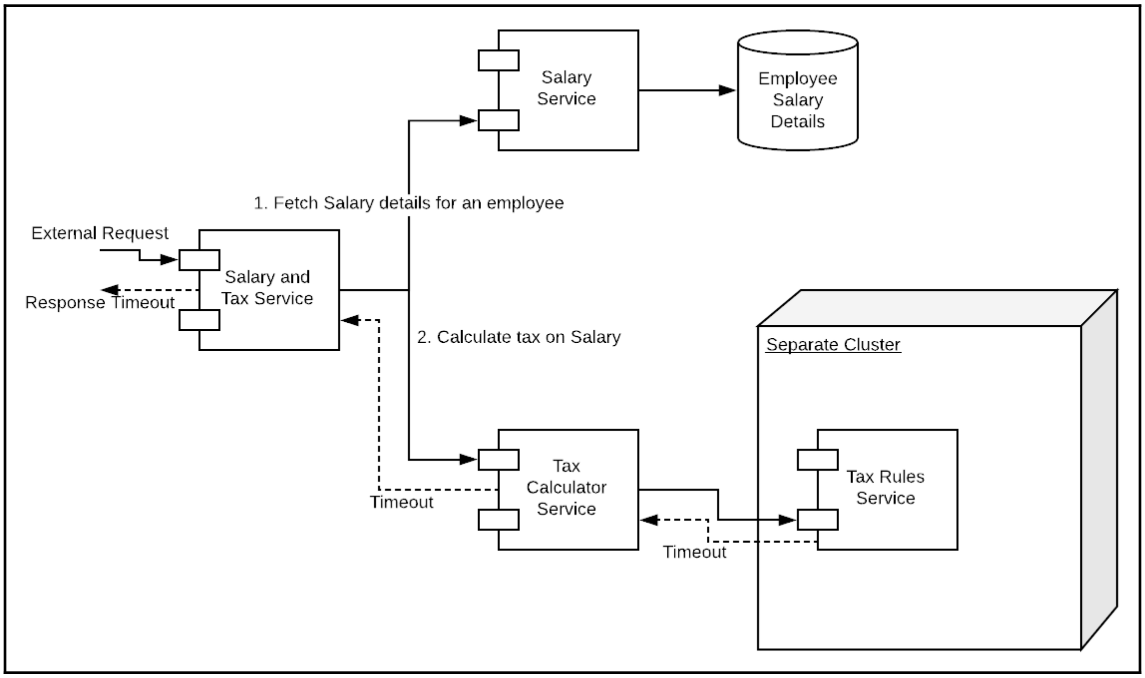
Let's take an example. Say we have a service that is supposed to return data about an employee's tax details. This service fetches some tax rules from another service and then works on the salary details provided. After calculating the tax details, the service returns the tax to be applied on the salary.

The following diagram highlights how tax and salary services interact:

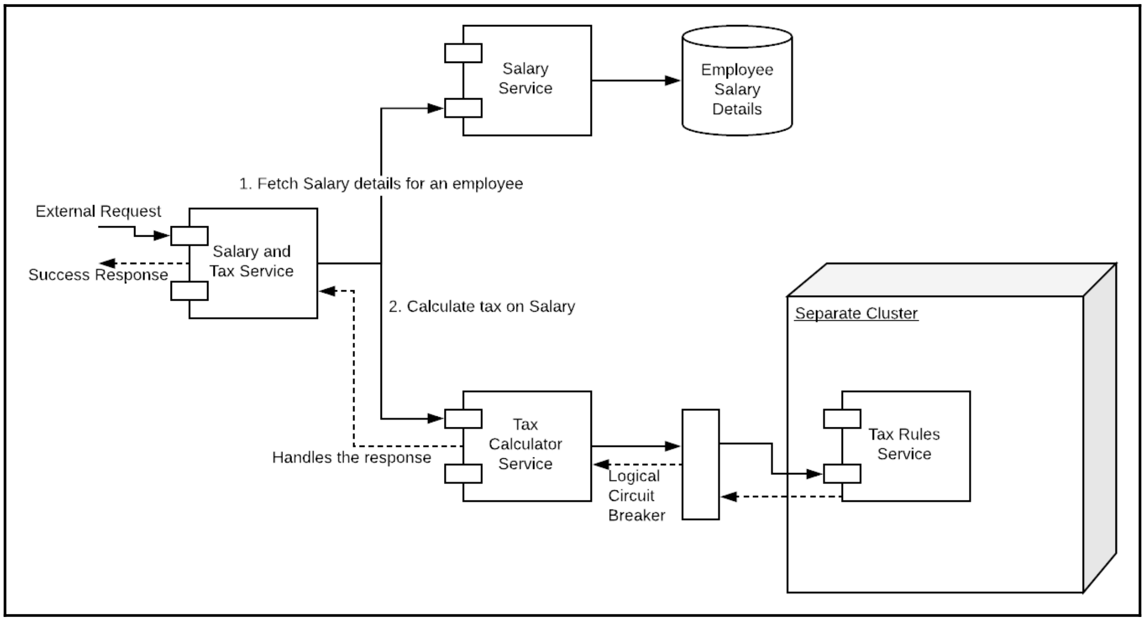


Now, let's say the **Tax Rules Service** stops responding; it might be dealing with a huge load or some code error. This would mean that our **Tax Calculator Service** would call the **Tax Rules Service** but will keep on waiting until it errors out, so we can see that a faulty service is actually impacting other services.

Both the calls will timeout, resulting in overall failure, as shown here:



A circuit breaker can help in this case by creating a protective layer across the services. This will help to break the connection between the calling service and the faulty service:



A circuit breaker would help us to isolate the faulty service.

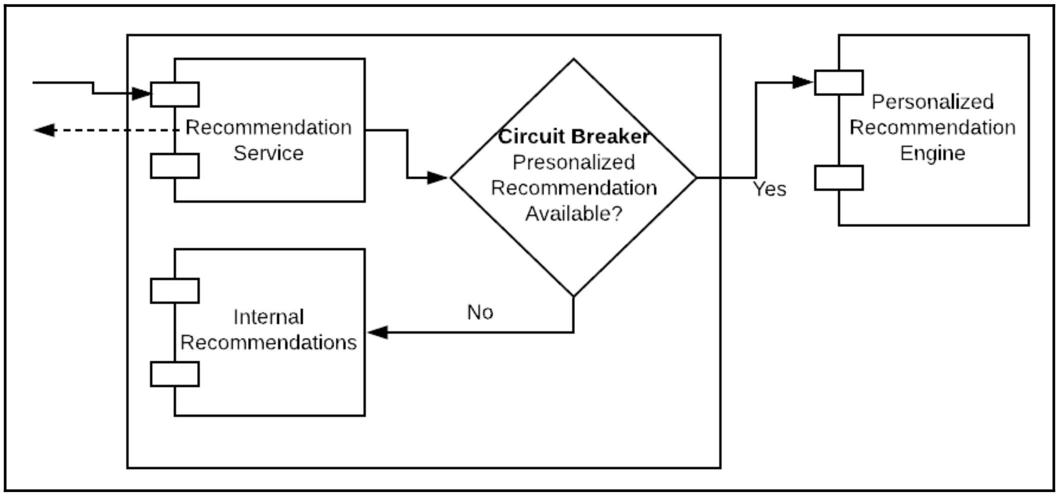
The circuit breaker can be in three states:

- **Closed**: When everything is working normally. At this time, the circuit breaker is in the closed state, all the calls are being made successfully to the service, and we are getting proper responses. We can add a threshold number for errors before the circuit breaker gets into the open state.
- **Open**: As the name suggests, the circuit is open or we are in an error state, so none of the calls will be made to the service being called.
- **Half-open**: Once our circuit goes into the open state, the circuit breaker will keep a check on the service being called to make sure things work normally once the service is healthy again. To do so, the circuit breaker gets into the half-open state after staying for a predefined period of time in the open state. In the half-open state, the circuit breaker will try calling the end service again; if the call succeeds, the circuit will go back to the closed state; otherwise, it will go into the open state again.

A circuit breaker code example

As you may have guessed, the circuit breaker is a very important pattern when it comes to handling service failures. It is worth spending some more time understanding how to implement it. Let's take a hypothetical problem; say we have a movie recommendation service. Users will make a call to our service, which in turn gets the recommendation for the current user from a sophisticated machine learning implementation. In case the external recommendation service goes down, we still want our users to be able to get recommendations.

We will try implementing the following solution:


Recommendation Service
Circuit Breaker
Presonalized Recommendation Available?
Yes
Personalized Recommendation Engine
No
Internal Recommendations

To get started, we will create a couple of Spring Boot applications, as will do in the rest of the book. Here is the recommendation engine service:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

/***
* This class is responsible for applying machine learning algorithm for
* the current user and find out personalized algorithm.
*
*/
@RestController
public class RecommendationController {

  private final RestTemplate restTemplate;
  public RecommendationController(RestTemplate restTemplate) {
    this.restTemplate = restTemplate;
  }

  /***
  * This method calculates and return movies recommendation
  * @return Movies
  */
  @GetMapping("/movies")
  public String getRecommendedMovies() {
  // Applies number cruncher algo and come up with movies for logged in
used
  // For sake of thie example we are returning fixed set of movies
    return("Movies Recommended for you. \n 1. Jumanji: Welcome to the
Jungle. \n2. Inception \n3. The Dark  knight.");
  }

}
```

But we are more interested in the calling method, where we will implement our circuit breaker. We will include a Netflix Hystrix implementation to use the circuit breaker:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

And here is the controller API and `service` class that implements the circuit breaker:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.client.RestTemplateBuilder;
import
org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.context.annotation.Bean;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@EnableCircuitBreaker
@RestController
@SpringBootApplication
public class ServiceoneApplication {

  @Autowired
  private RecommendationService recommendationService;
```

```
    public static void main(String[] args) {
      SpringApplication.run(ServiceoneApplication.class, args);
    }

    @Bean
    public RestTemplate rest(RestTemplateBuilder builder) {
      return builder.build();
    }

    @RequestMapping("/movies")
    public String getRecommendedMovies() {
      return recommendationService.recommend();
    }
  }
```

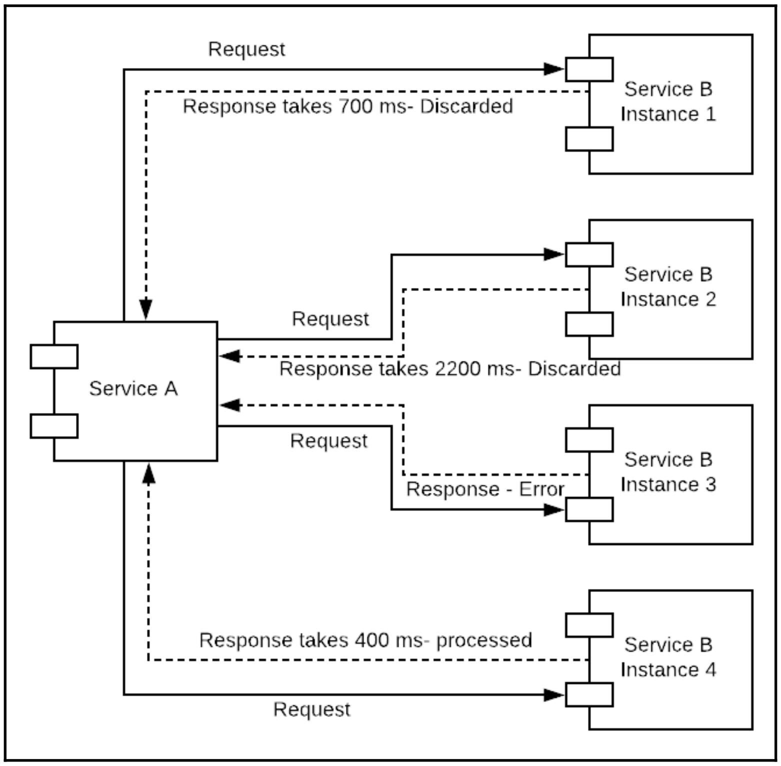
The following is the for the recommendation service:

```
  import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
  import org.springframework.stereotype.Service;
  import org.springframework.web.client.RestTemplate;

  import java.net.URI;

  @Service
  public class RecommendationService {

    private final RestTemplate restTemplate;

    public RecommendationService(RestTemplate rest) {
      this.restTemplate = rest;
    }

    @HystrixCommand(fallbackMethod = "reliable")
    public String recommend() {
      URI uri = URI.create("http://localhost:8081/movies");
      return this.restTemplate.getForObject(uri, String.class);
    }

    public String reliable() {
      return "Top 3 Movies for the month. \n 1. Batman Begins\n 2.
  Interstellar\n 3. Justice League. ";
    }

  }
```

Fan out and fastest response

This approach is a bit costly in terms of computing power, but is useful in case response time is critical for our application. The idea in this approach is to have multiple replicas of our service. The caller will simultaneously call all these instances and accept the response from the one that responds fastest. As there is additional overhead in the use of this approach, caution is recommended.

Let's take a look at an example scenario:



The example given here shows that **Service A** needs to call **Service B** and fetch a response. Calling the service would send simultaneous requests to multiple instances of services being called. The response received fastest will be processed by the calling service. We can see the advantage of this approach is that if some instances are in an error state or taking too much time to respond, it will not impact our service as long as any one of the instances being called is able to send the response in a timely manner. As already mentioned, this approach adds to the cost of the overall system in terms of a greater hardware power requirement, so caution is required when using this approach.

Recovering from failure

So far, we have discussed isolating the failure and handling it. So now we are handling our failures gracefully, but what about fixing them? We cannot let the system remain in an error state forever. We will need to bring back our filed services and make sure our system is in a healthy state again.

Well, how exactly we recover from failures will depend on the system. How much manual effort would be required to fix the issues? For example, we cannot of course fix the code issue on the fly. But, we can take some steps to help and speed up the recovery process.

The most important tool we have for recovery from a failure is monitoring. Proper monitoring of our services would let us know about the services facing issues. We need to monitor whether a service is responding correctly or with error codes. We need to monitor logs for exceptions and errors. We need to monitor the hardware health of the nodes, such as memory and CPU usage. If the services are throwing too many errors, above an acceptable threshold, or other parameters such as memory or CPU usage are beyond acceptable levels, the monitoring script can trigger an action. An action could be as simple as raising a trigger to concerned teams by sending emails, messages, or escalations to restart nodes. The automated scripts can also take a call on whether we need to scale up the service by adding additional nodes. We will talk more about monitoring and scaling in chapters dedicated to these topics. `Chapter 8`, *Monitoring Microservices* and `Chapter 6`, *Scaling Microservices*.

Preparing for failure

So, we have talked about handling and fixing failures. They say the more you sweat in peace, the less you bleed in war. So, it is a good idea to keep yourself prepared for any expected or unexpected failures.

We need to make sure we have a recovery process in place. A simple example is that when we are moving to a newer version of a service, it will make sense to keep the older version live for some time so that if something goes wrong, we can quickly move back to an older version.

Netflix, which relies heavily on Microservices for their system to work properly, came up with the approach of Chaos Monkey. The idea is simple, yet powerful: randomly bring down instances of services in place and see how your system behaves. Does it completely shut down or is it able to handle the situation? Is the service be able to heal itself? Is the system be able to take the required actions to bring it into a healthy state?

This way, we will can determine our weaknesses and take the required actions in time.

Summary

In this chapter, we talked about how to make our microservices-based application reliable. We discussed different aspects, such as isolating, handling, recovering from, and preparing for failure.

For isolating failure, we explored techniques such as the bulkhead pattern, stateless services, and the Robustness Principle. When discussing Handling failure, we touched upon the approaches of Asynchronous communication, Fail fast, Timeouts, circuit breakers, and fan out and fastest response.

We talked about the importance of monitoring logs and hardware performance when we discussed recovering from failure. Finally, we talked about steps such as simulating production failures so that we are better prepared for a failure situation.

In the next chapter, we will discuss about the Scaling of Microservices.

6
Scaling Microservices

An important aspect that has made Microservices popular is the ease with which they can be scaled. In this chapter, we will study the scalability aspect of Microservices. We will look at what makes Microservices the best fit for applications that need quick scalability. The following topics will be covered in this chapter:

- What is scalability?
- Microservices and scalability
- Stateless versus Stateful Microservices
- Microservices on the cloud
- Going serverless with Microservices
- Scaling databases with Microservices
- Scaling Microservices with caching

What is scalability?

Has this ever happened to you:, you created an application that worked well in a controlled test environment but when it got deployed to the production environment, parts of the application stopped responding? Or the application works well for the first few days, when the load or number of users is low, but as the load increases, the application stops responding or slows down? These are very common scenarios if you created an application without thinking about the scalability aspect of it.

So, what exactly is scalability?

Scalability can be defined as the ability to scale your operations as per increases in demand. With respect to a computer system, it would mean the application adapting to increased load. I would say this definition is simple, crisp, and covers the gist, but it is still a bit incomplete. The explanation given for scalability talks about handling increased demand, but what about decreased demand? Let's hold on to that thought. We will come back to it later but first, let's consider a case of increased demand and scalability.

Let's take a case; there is a website that handles 1,000 users daily. We have a fixed Linux server of a specific capability that is happily handling this load. Let's leave out the database, application server, other tools, and technology-related complexity for now. Assume that slowly and steadily, the website is getting popular and the number of users doubles. But we realize the machine we have deployed it on is not able to handle this load and chokes. You could handle the situation by adding more power to the machine or moving to an upgraded one. Say we used 8 GB of RAM earlier and we moved to 16 GB RAM, that would solve our problem.

Next, we see an increase in load and we again double the capacity. This is called **vertical scaling**, which is where we add more power to our machine and the same instance of the application handles more load. This is good, but requires manual effort and is not cost-effective. We know that having a high-end server machine is costlier than having two low-configuration machines. To take advantage of this concept, we would like to be able to add more instances of our application and get better performance than by adding more power to the same machine. This is called **horizontal scaling**.

So, for the purpose of this book, we will define scalability as the ability of the system to adapt to increased or decreased demand, with the optimum use of resources available.

Note that I am also emphasizing a decrease in demand. This is one aspect that can be neglected when we talk about scalability. This is understandable, as everyone would like to think of scenarios where the applications are becoming popular and the load is increasing. But in the real world, we have to deal with temporary loads at times; for example, an e-commerce site would see a heavy load for an online sale period. That's why, when I started discussing scalability, I mentioned that you should architect for scaling down along with scaling up. At the end of the day, we do not want to pay for resources that we are not using.

Microservices and scalability

At the start of the chapter, I mentioned Microservices are a natural fit for scalable applications. Well, it is always easier to make copies of a smaller piece of code than a huge, monolith application, and as we have seen so far in this book, Microservices is all about breaking a monolithic application into smaller and more meaningful services.

Most importantly, Microservices can scale independently. Let's take an example: we have a huge e-commerce application that has multiple services, such as search, payment gateway, product catalog, and shopping cart. We are observing a lot of load on search due to the launch of new product lines. It is easier to just create replicas of search services and let the search functionality scale independently without worrying about the rest of the services. Here's another example: on a college admission day, we see a lot of load on the admission form-submission service, whereas the rest of the services, such as attendance or exam scheduling are seeing an almost negligible load. We can upscale the admission service without impacting the rest of the services.

Due to this independent nature of services, Microservices are a good fit for environments where one or more services is required to scale independently, without impacting the rest of the application.

Stateless versus Stateful scalability

We can write our services either in a stateful or stateless manner. I am referring to session state here. For example, at times we need to make sure that the user is logged in before we let them access a resource or perform an operation. Or we might want to maintain the previous state of operations; for example, you can submit form 2 only if you have submitted form 1.

To understand statelessness, let me show you a very vanilla service example. The following code shows a service to add up, two numbers:

```
package com.calculate.sum;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SumController {
  @GetMapping("/sum")
  public Integer addNumbers(@RequestParam Integer num1, @RequestParam
Integer num2) {
    return num1+num2;
  }
}
```

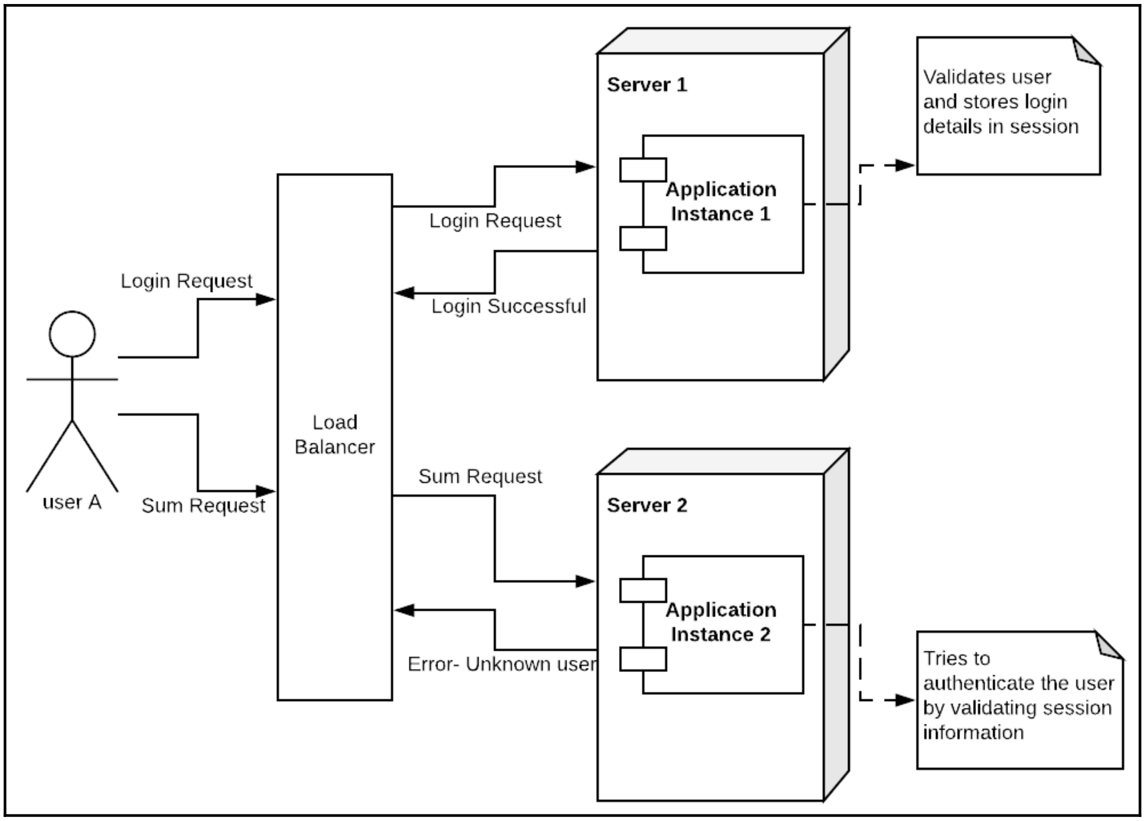
This is a simple Spring REST service that takes two parameters, `num1` and `num2`, and returns the `sum`. This is a completely stateless service, which is easy to scale. It is stateless because this service does not care about who is requesting, what call was made before and after this service was called, and so on. All it does is take an input and return the output.

Why do we say it is easy to scale? Well, I can add dozens of instances of the service behind a load balancer URL, it doesn't matter which instance serves the request as the result will always be same.

But, let's say we have a condition that only logged in users, that is, authenticated users, should be able to access this `sum` service. This adds complexity as now we somehow need to check that the user is indeed logged in before serving the sum. Traditionally, if we are dealing with a simple application getting deployed on a single server, an easy way out is the use of sessions:

```
package com.serviceone.serviceone;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SumSessionController {

  @GetMapping("/sum2")
  public Integer addNumbers(HttpServletRequest req, @RequestParam Integer
num1, @RequestParam Integer num2) {
    HttpSession sess = req.getSession(false);
    if(sess!=null) {
      Boolean authVal = (Boolean) sess.getAttribute("isAuthenticated");
      if(authVal)
        return num1+num2;
    }
    return -9999;
  }
}
```

In the preceding code, we are making sure only an authenticated user is able to use the service. Sessions are maintained at the server level. The problem with this implementation is that it is hard to scale. For example, say we added multiple instances of the application on different servers. Now the initial login request, where we are setting the session information after a successful login, goes to **Server 1**. And after that, a subsequent request goes to **Server 2**, which does not have any details on user login (as that information is with **Server 1**), so it will assume the user is not authenticated and throw an error:



The problem with this implementation is that we need to make sure that all the requests from the same user are received by the same server instance.

There are solutions, such as sticky sessions, that would make sure all the requests coming from one IP address go to the same server, which would solve our problem. But this is not a perfect scenario as we are adding the limitation that all the requests from one user need to go to the same server, so we will not be able to make sure we are evenly distributing the load to our servers.

To solve this problem, we can use token-based authentication. Let's take an example of an implementation that is based on `JWT` to make it clear:

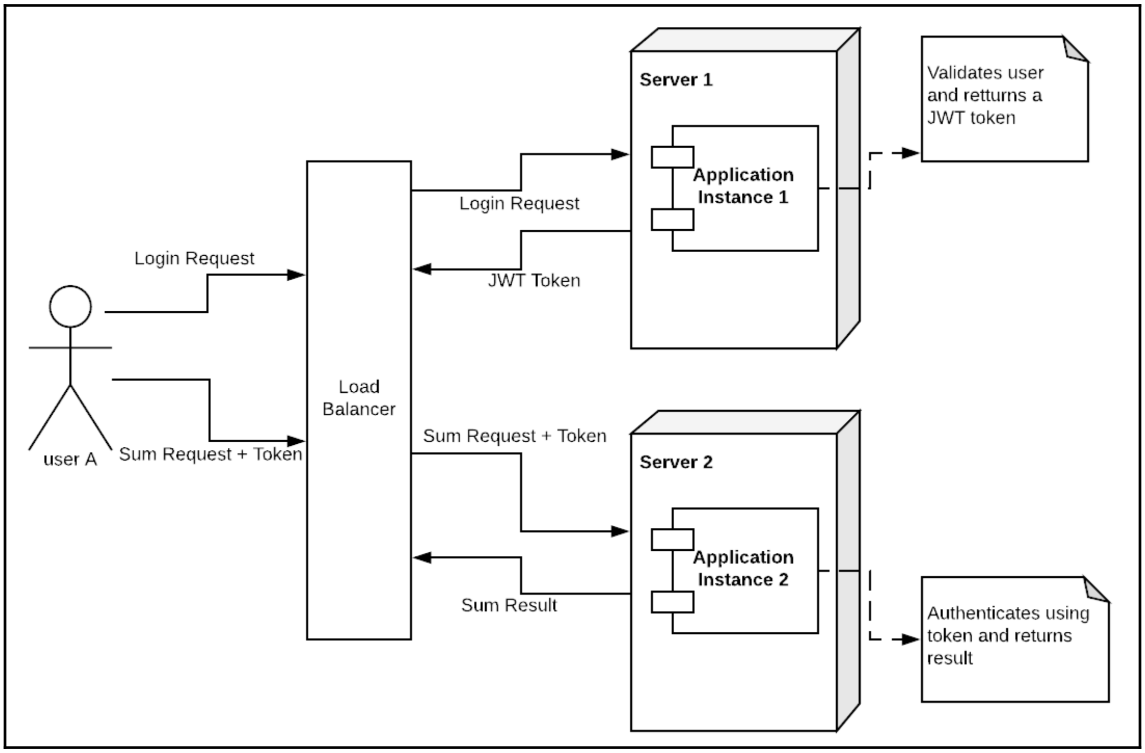
```
package com.statemanager;

import java.util.Calendar;
import java.util.Date;

import com.auth0.jwt.JWT;
import com.auth0.jwt.JWTCreator;
import com.auth0.jwt.JWTVerifier;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.interfaces.DecodedJWT;

public class JWTUtil {
// You will use a application secret stored in properties file or database.
s
String secret = "KeepSecretKeySameAccrossServers";

  /**
  * This method takes a user object and returns a token.
  * @param user
  * @return
  */
  public String createAccessJwtToken(User user) {
    Date date = new Date();

    Calendar c = Calendar.getInstance();
    c.setTime(date);
    c.add(Calendar.DATE, 1);
    // Setting expiration for 1 day
    Date expiration = c.getTime();
    JWTCreator.Builder builder = com.auth0.jwt.JWT.create();
    builder.withSubject(user.getName())
      .withKeyId(user.getId())
      .withIssuedAt(date)
      .withExpiresAt(expiration)
      .withClaim("canAccessSum", true);
    String token = builder.sign(Algorithm.HMAC256(secret));
    return token;
  }

  /**
  * This method takes a token and returns User Object.
  * @param token
  * @return
  */
  public User parseJwtToken(String token) {
    JWTVerifier verifier = JWT.require(Algorithm.HMAC256(secret)).build();

    DecodedJWT jwt =verifier.verify(token);

    Boolean sumAccess = jwt.getClaim("canAccessSum").asBoolean();

    User user = new User();
    user.setId(jwt.getId());
    user.setName(jwt.getSubject());
    user.setSumAccess(sumAccess);
    return user;
  }
}
```

The preceding code helps us generate a token that can keep all the mandatory information required to serve the user. We can pull the required information at runtime from the token, so there is no need to maintain internal state and store user data on the server as part of session information.

The following code showcases the use of a JWT token:

```
package com.statemanager;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SumSessionController {

  @GetMapping("/sum2")
  public Integer sayHello(@RequestParam String token, @RequestParam Integer
num1, @RequestParam Integer num2) {
    User user = new JWTUtil().parseJwtToken(token);
    if(user.getSumAccess()) {
      return num1+num2;
    }
    return -9999;
  }
}
```

This helps us keep services stateless and hence independently scalable:



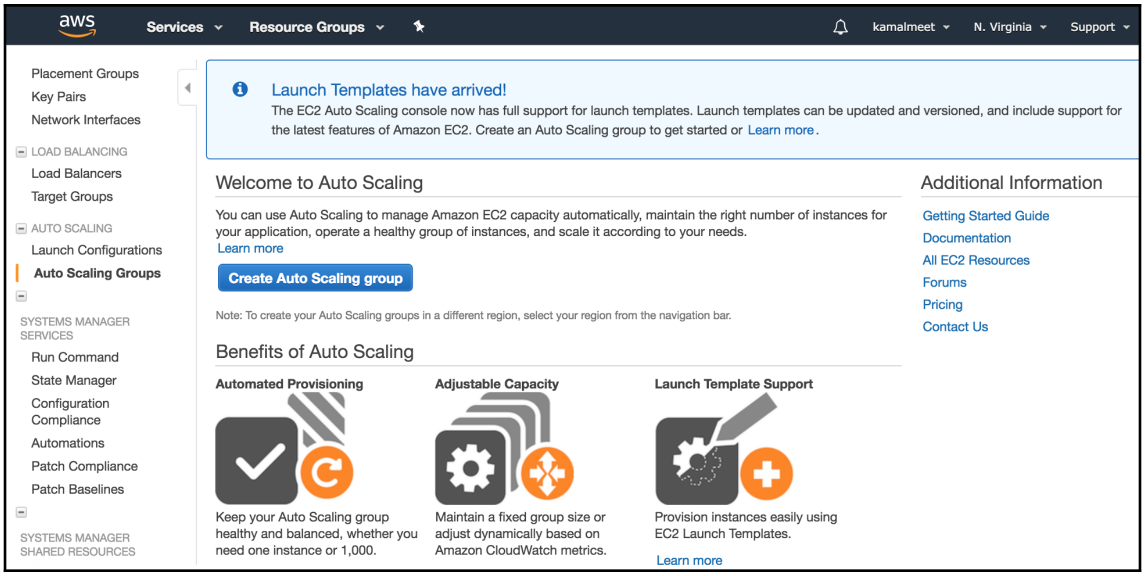
We can see how tokens can help us keep our services session free, so there is no need to store information in server-side sessions. At the same time, we make sure that we can pass state information along with each request. We will look at JWT again in `Chapter 7`, *Securing Microservices,* where we will focus on the security aspect of tokens.

Scaling on the cloud

With the popularity of the cloud, scaling Microservices has become even easier. In fact, it would not be completely inappropriate if we say that the cloud is one major reason for the popularity of Microservices based design. Cloud services providers, such as **Amazon Web Services** (**AWS**), provide us with mechanisms to autoscale the deployed services.

For example, let's see how this is done in AWS. If you are familiar with Microsoft Azure, Google Cloud, or any other cloud service provider, you will find similar steps there. We are just trying to get a high-level idea here, for which I am using AWS.

Let's say you have created a set of **Elastic Cloud Compute** (**EC2**), or simply a virtual machine instance, that is servicing a Microservice or set of Microservices. You can create an Auto-scaling Group that helps us to set a minimum number of instances available for the service, or group of services, that is deployed:



Launch Templates have arrived!

The EC2 Auto Scaling console now has full support for launch templates. Launch templates can be updated and versioned, and include support for the latest features of Amazon EC2. Create an Auto Scaling group to get started or Learn more.

Welcome to Auto Scaling

You can use Auto Scaling to manage Amazon EC2 capacity automatically, maintain the right number of instances for your application, operate a healthy group of instances, and scale it according to your needs. Learn more

Note: To create your Auto Scaling groups in a different region, select your region from the navigation bar.

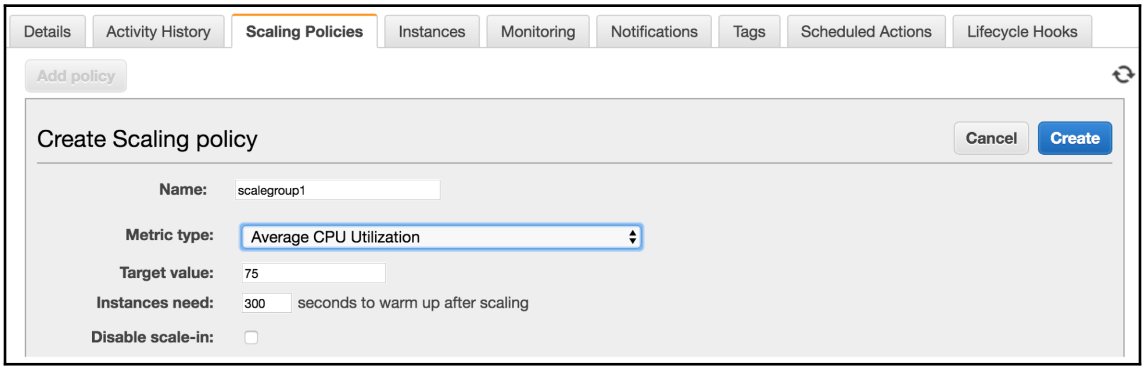
Benefits of Auto Scaling

Automated Provisioning: Keep your Auto Scaling group healthy and balanced, whether you need one instance or 1,000.

Adjustable Capacity: Maintain a fixed group size or adjust dynamically based on Amazon CloudWatch metrics.

Launch Template Support: Provision instances easily using EC2 Launch Templates. Learn more

Next, all you need to do is set a rule for the creation of more instances. In short, we can autoscale based on rules, and no human interaction is required:



Create Scaling policy

Name: scalegroup1

Metric type: Average CPU Utilization

Target value: 75

Instances need: 300 seconds to warm up after scaling

Disable scale-in: ☐

Going serverless with microservices

If you're working in the software industry, you've probably already heard of serverless services and design. Serverless architecture takes the Microservices-based approach to the next level. With the use of serverless architecture, we are completely moving away from managing any hardware. This also means that we do not need to worry about our scalability needs as scaling up or down will be taken care of by the service provider.

Let's take a quick look at how can we create a serverless implementation of a Microservice that can be used to sum up any two numbers. For the sake of this example, we will use the AWS Lambda serverless implementation, but you can use equivalent services in Microsoft Azure, Google Cloud, or other providers.

First, we will create a class file for the Lambda function's implementation:

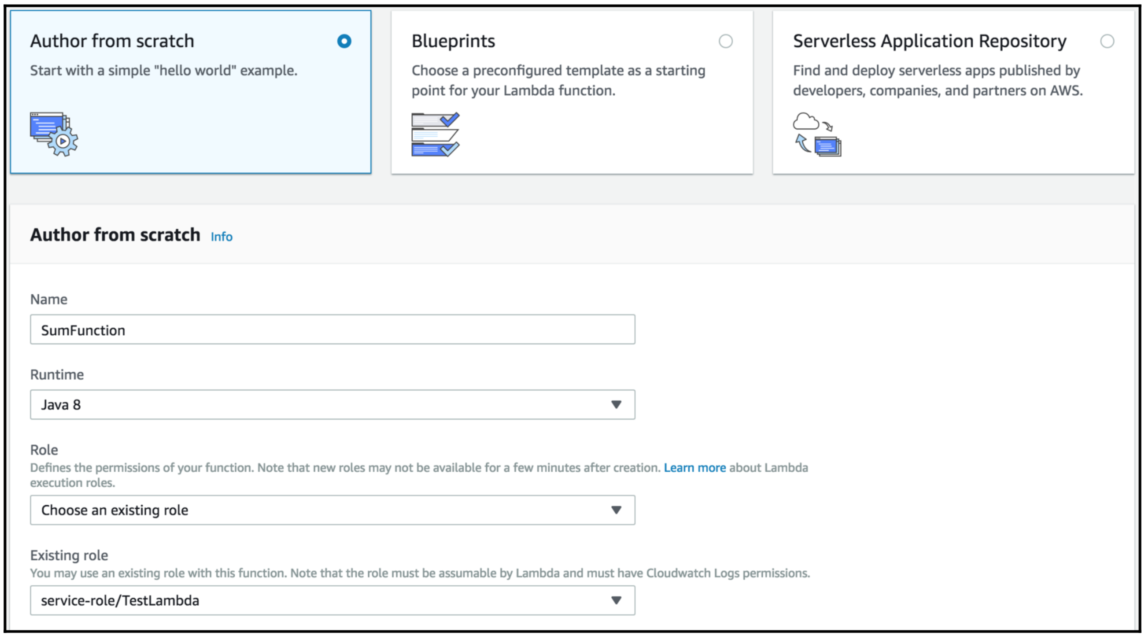
```
package com.test;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;

import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestStreamHandler;

/**
* Class to implement simple hello world example
*
*/
public class LambdaMethodHandler implements RequestStreamHandler {

  public void handleRequest(InputStream inputStream, OutputStream
outputStream, Context context) throws  IOException {

    BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
    JSONObject responseJson = new JSONObject();
    int num1 =0;
    int num2 = 0;
    String responseCode = "200";
    try {
      // First parse the request
      JSONParser parser = new JSONParser();
      JSONObject event = (JSONObject)parser.parse(reader);

      if (event.get("queryStringParameters") != null) {
        JSONObject queryStringParameters =
(JSONObject)event.get("queryStringParameters");
        if ( queryStringParameters.get("num1") != null) {
          String temp = (String)queryStringParameters.get("num1");
          num1 = Integer.parseInt(temp);
        }
        if ( queryStringParameters.get("num2") != null) {
          String temp = (String)queryStringParameters.get("num2");
          num2 = Integer.parseInt(temp);
        }
      }

      // Prepare the response. If name was provided use that else use
default.
      String sum = "Sum is " + (num1+num2);

      JSONObject responseBody = new JSONObject();
      responseBody.put("message", sum);

      JSONObject headerJson = new JSONObject();
      responseJson.put("isBase64Encoded", false);
      responseJson.put("statusCode", responseCode);
      responseJson.put("headers", headerJson);
      responseJson.put("body", responseBody.toString());

    } catch(ParseException parseException) {
      responseJson.put("statusCode", "400");
      responseJson.put("exception", parseException);
    }

    OutputStreamWriter writer = new OutputStreamWriter(outputStream,
"UTF-8");
    writer.write(responseJson.toJSONString());
    writer.close();
  }

}
```

Next, we will execute the build using the following command:

```
mvn clean package shade:shade
```

Go to Amazon Lambda. From there, select the AWS account, choose **Lambda service** | **Create function** | **Author from scratch,** and provide the required values.

The following screenshot shows the screen from AWS Lambda:



Author from scratch ● Start with a simple "hello world" example.

Blueprints ○ Choose a preconfigured template as a starting point for your Lambda function.

Serverless Application Repository ○ Find and deploy serverless apps published by developers, companies, and partners on AWS.

Author from scratch Info

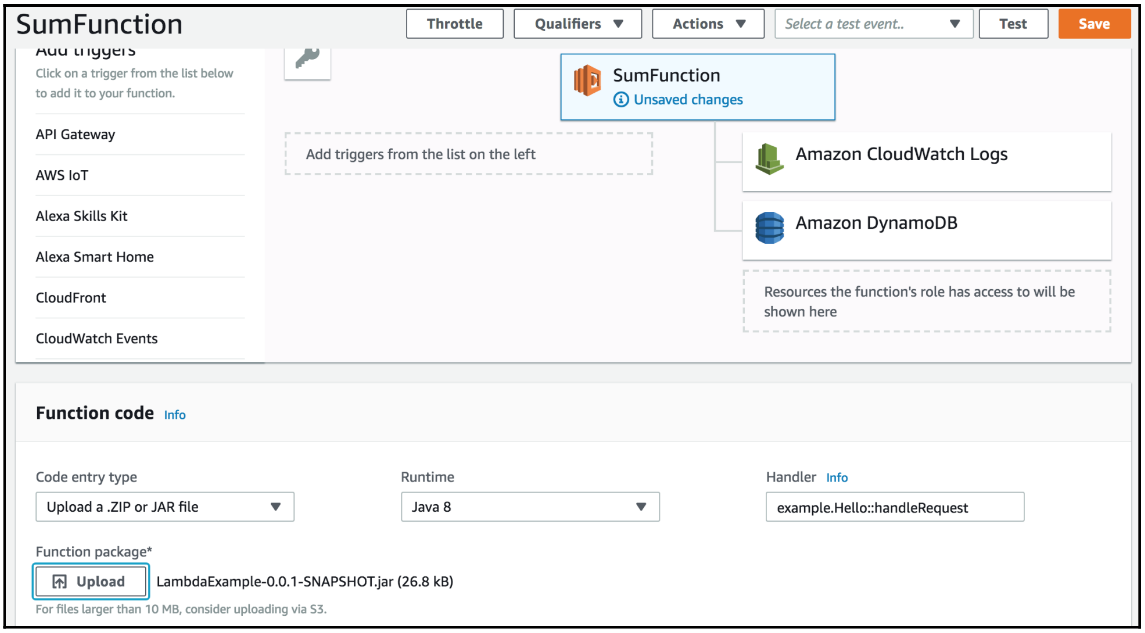
Name
SumFunction

Runtime
Java 8

Role
Defines the permissions of your function. Note that new roles may not be available for a few minutes after creation. Learn more about Lambda execution roles.
Choose an existing role

Existing role
You may use an existing role with this function. Note that the role must be assumable by Lambda and must have Cloudwatch Logs permissions.
service-role/TestLambda

Next, it will ask us to upload the `.Jar` file we created for our sum function:



Code entry type: Upload a .ZIP or JAR file

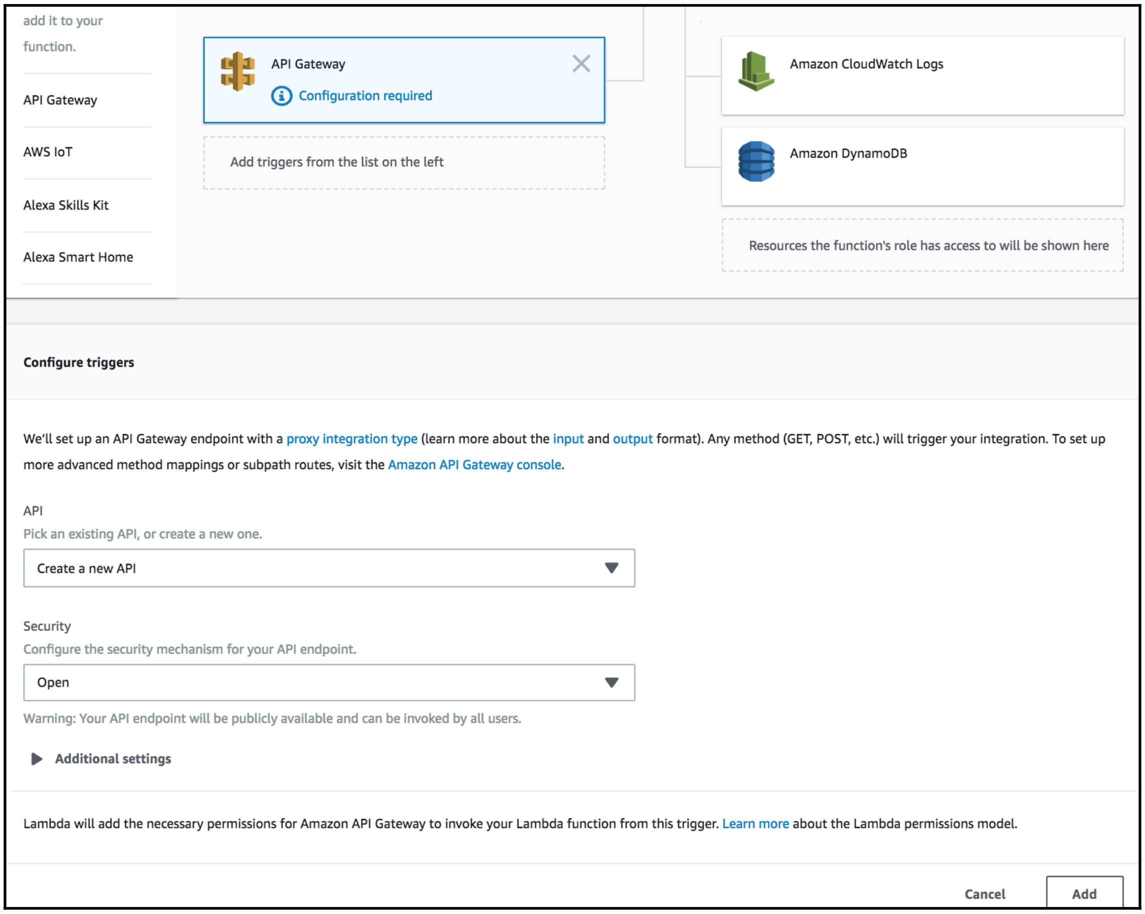
Runtime: Java 8

Handler: example.Hello::handleRequest

Function package*: LambdaExample-0.0.1-SNAPSHOT.jar (26.8 kB)

Update the `handler` function, under the *Handler* section. For example, in this case, provide `com.test.LambdaMethodHandler::handleRequest`.

After uploading `.Jar`, the last step is to configure the API gateway. The following screenshot shows the configuration screen:



add it to your function.

Configure triggers

We'll set up an API Gateway endpoint with a proxy integration type (learn more about the input and output format). Any method (GET, POST, etc.) will trigger your integration. To set up more advanced method mappings or subpath routes, visit the Amazon API Gateway console.

API
Pick an existing API, or create a new one.
Create a new API

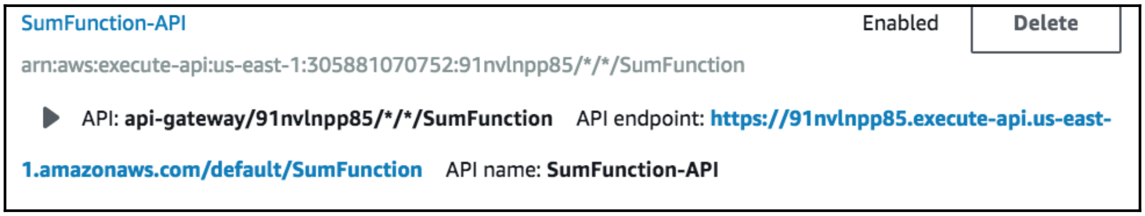
Security
Configure the security mechanism for your API endpoint.
Open
Warning: Your API endpoint will be publicly available and can be invoked by all users.

Additional settings

Lambda will add the necessary permissions for Amazon API Gateway to invoke your Lambda function from this trigger. Learn more about the Lambda permissions model.

After filling in the details, we will be provided with the URL,which can be used to call the Microservice we created.

The following screenshot shows the API URL provided by AWS:



SumFunction-API

Enabled

Delete

arn:aws:execute-api:us-east-1:305881070752:91nvlnpp85/*/*/SumFunction

API: **api-gateway/91nvlnpp85/*/*/SumFunction** API endpoint: **https://91nvlnpp85.execute-api.us-east-1.amazonaws.com/default/SumFunction** API name: **SumFunction-API**

In this example, AWS provided us the following URL to call our service:

```
https://91nvlnpp85.execute-api.us-east-1.amazonaws.com/default/SumFunct
ion?num1=1&num2=4
```

The advantage of this approach is that we are not worried about where the service is deployed, or how to scale up or down. The cloud service provider will automatically take care of all our scaling needs. In this model, you only need to pay for the infrastructure you are using; in other words, you will be charged based on the number of calls your Microservice is receiving.

Though a serverless based approach looks very good, you need to be careful when selecting a solution approach. One needs to look at the pros and cons of any solution being finalized. The serverless approach might have some limitations from the service provider end, for example, currently Amazon has a limitation that a service should run in a maximum of five minutes.

Scaling databases with Microservices

One of, Microservices rules is that every service is supposed to manage its own data, which makes it easy to scale data. The idea is that managing and handling a smaller set of data is easier than managing a monolithic database. For example, in our example of an e-commerce application, say we have one product service that manages product-related information, another for users, and so on. The first thing to note is that we are not trying to keep all the data in one database machine. Each service can scale independently with its own data.

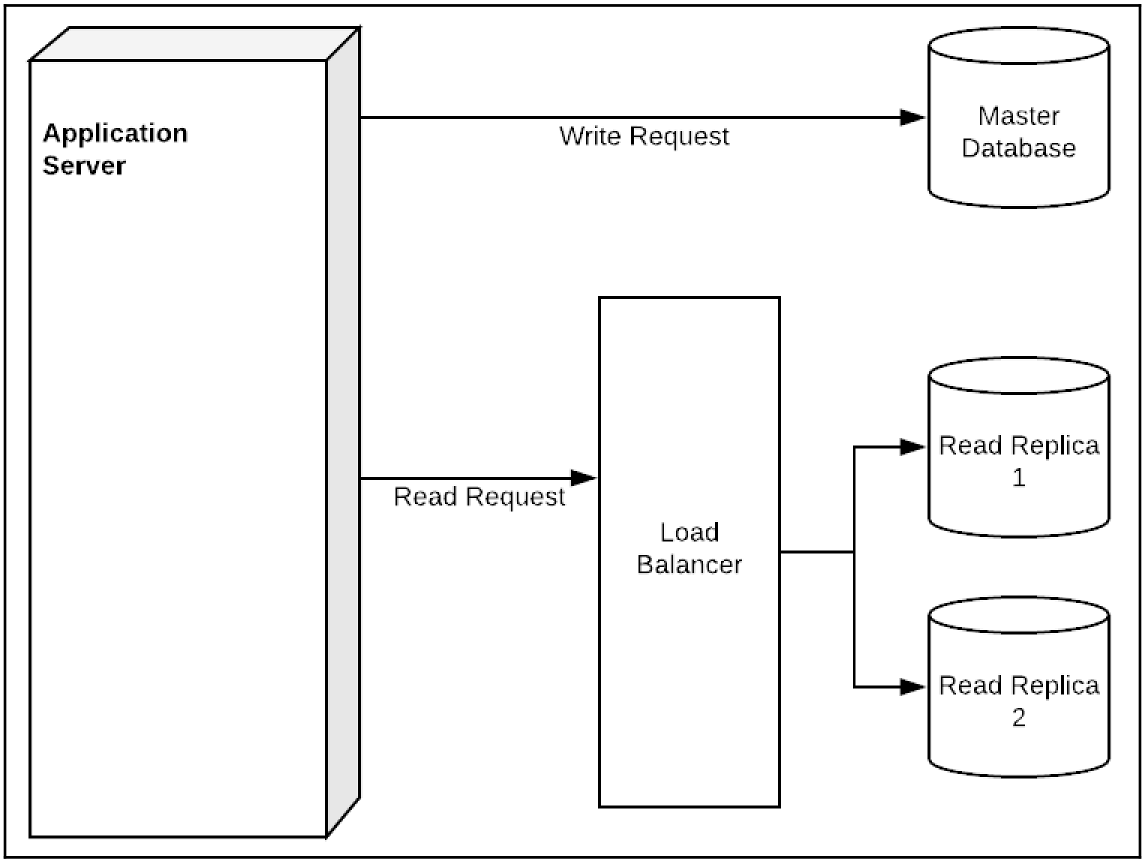
How you manage your data will depend on the kind of application you are trying to build. But it is worth spending some time understanding the scalability aspects of a database. Traditionally, when designing for scalability, databases were the most difficult part of an application to scale and used to usually be the bottleneck. I remember in the early days of my career – when NoSQL (we will come to this soon) databases were not so popular, and we used to rely more on Relational or SQL databases – scaling was a nightmare. It wasn't impossible, but it was definitely tricky. Let me explain what I'm talking about.

Imagine we have this e-commerce application that has different important aspects, such as products (details of products being sold), inventory, users, shopping cart, orders, and transactions. Everything works fine until we have a few thousand products with a few thousand users. Imagine the site starts getting more popular, which of course we want, and the number of products and users is getting into the millions. We had our relations database on 100 GB of storage, which is no longer sufficient. Ages ago, the obvious solution was to move our database to a bigger machine, with more disk, memory, and computational power. But, as already discussed, this approach has its limitations. A single high-power machine is usually costlier than multiple low-power machines. And, more importantly, there is a limit to increasing the power of a single machine; that is, you can add only a limited amount of CPU or RAM to a system.

There are solutions to this problem; two of the most common are read replicas and sharding.

Read replicas are a solution where we create additional database imagread-only only copies. The replicas would only serve read requests. This solution usually helps in cases where we expect more read operations than data manipulation operations. For example, when people are browsing products on a website, they are actually not modifying any data. The idea is to distribute these search requests across read-only database images.

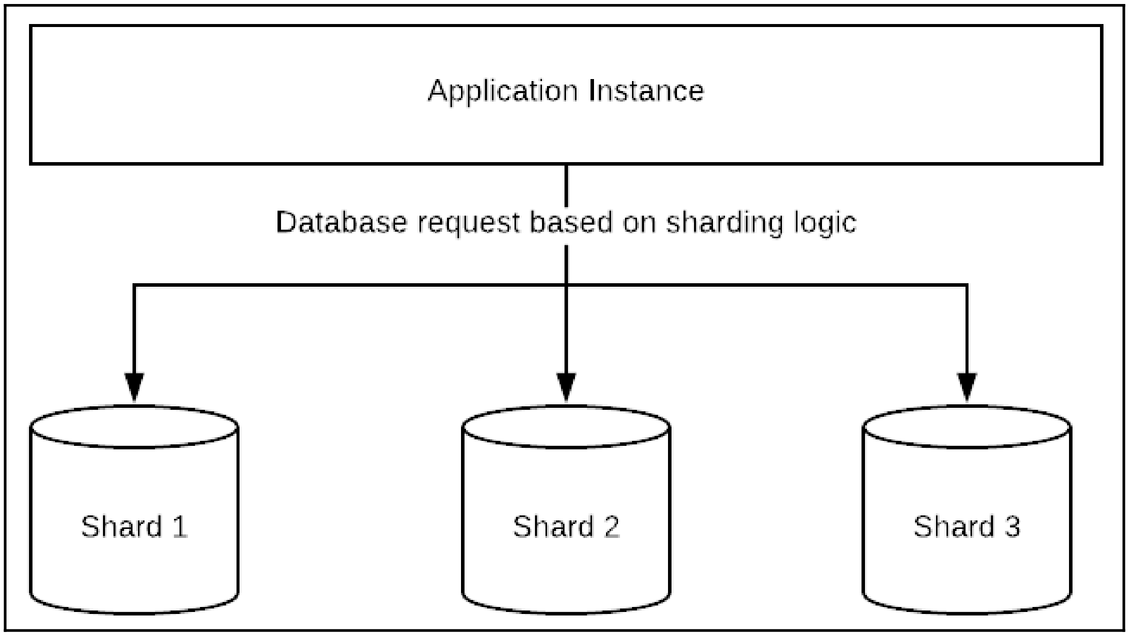
The following diagram shows the use of read replicas:



Though read replicas help to speed up fetch queries, the design is not without its drawbacks. For example, if we added a new product or an existing product goes out of stock, the information first gets updated to the main database and then copied over to read replicas. There is some lag in this operation; copying from the main database to replicas will take some time. The delay can last anywhere from a few seconds to a few minutes, based on the business logic applied. So while this data is getting updated, users are seeing stale data. Imagine the frustration of users who saw a product, added it to their cart, but when they were checking out they got a message that the product was out of stock.

Another important and useful aspect that helps to scale a database is sharding. The idea behind the concept of sharding is to logically divide data into multiple databases, where it helps to keep the data in different machines. A simple example would be in that the case of an e-commerce site, we would like to keep data from different countries in different database instances. Another common way to divide data is through a primary key or some other logical divider. For example, in a database that manages employee data, we might want to keep data for employees whose surnames start from A-H in one database, I-P in another, and Q-Z in another.

The following diagram shows how a sharded database appears:



Though sharding is another important and widely-used approach for scaling databases, it comes with its own problems. The most common one is joining the tables. In a single database instance, it would be easy to join multiple tables and generate reports, whereas if data for one table is in one machine and another table is in the second machine, it is difficult to join the two tables and fetch any meaningful data.

NoSQL databases have given us many options in managing database scalability. Before going into detail, let me take a moment here to explain what NoSQL databases are, and how they help us manage scalability. I believe NoSQL itself is a somewhat misunderstood the concept. A lot of the time, I have heard of Not-SQL being referred to as something that would simply solve all your scalability problems automatically.

So, what is NoSQL? Literally, it would mean No-SQL. But I actually prefer the term non-relational database. Let's look into this concept. NoSQL actually is an umbrella term, which groups all the databases that do not keep data in traditional relational or table-based formats.

There are four major types into which we can classify NoSQL databases, based on the approach they use to store the data:

- **Key-value based database**: This is perhaps the simplest kind of database. We are storing data in the form of key-value combinations. Think of a hashmap kind of structure, where we are adding a unique identifier as a key and data objects as values. It is very easy to scale, as long as we have unique keys. Examples of this kind of database include Redis and Riak.
- **Column-based databases**: We have been using row-based relational databases for a long time. In row-based databases, we think of a record as a single object that can be stored in a database's table row. For example, an employee record that has the ID, name, department, salary, and joining date can be thought of as a single record of a relational database:

ID	Name	Department	Salary	Joining date
1211	Joe	Finance	70000	29-07-2014
1212	David	IT	77000	17-09-2016

Let's say most of our requirements are around grouping data based on different aspects, such as, fetch how many employees are in IT, how many employees earn more than $75,000?, or, find the average salary. Row-based storage is not very effective in such cases.

In contrast, a database with column-based storage stores data based on the columns. For example, think of all the salary data is present independently and can be accessed for independent calculation without worrying about what extra data like department, joining date etc is available. This makes the calculations and access much easier based on a particular column.

Examples of databases using a column-based storage mechanism are Cassandra and Vertica.

- **Document-based databases**: This can be thought of as an extension to key-value-based storage. In a key-value-based system, a value can be anything, but a document-based database adds a restriction for proper formatting to be followed for data being stored. The data that is stored as documents. Metadata is provided for each document being stored, for better indexing and searching. Examples of document-based storage databases are MongoDB and CouchDB.
- **Graph-based databases**: This kind of database is useful when our data records are connected to other records in some way and we need a method to parse this connectivity. A simple example is when we are storing information for people, and we need to capture friendship information, such as P1 is a friend of P2, who in turn is a friend of P4 and P6, so we can capture some relationship between P1, P2, P4, P6, and so on. Examples of databases that use graph-based storage are Neo4J and OrientDB.

A complete discussion on databases is outside the scope of this book, so we will quickly jump back to our core topic, the scalability of Microservices and databases.

As we have already mentioned, with a Microservices-based architecture, it is recommended that each Microservice be responsible for its own data. The first advantage we get with this approach is that we are not dealing with a huge set of data in one go.

The second important advantage we get is that because each Microservice is independent, it is easy to manage different types of data stores. For example, you might want to keep product data in a document-based database, such as MongoDB, whereas you keep user information in traditional relational databases, such as Oracle RDBMS or MySQL.

Command Query Responsibility Segregation (**CQRS**)**:** We are used to looking at any entity as having four core operations, known as CRUD operations. These are Create, Read, Update, and Delete. Normally, we would have a single service, model, and database managing all these operations, but sometimes it makes sense to segregate Command (Create, Update, and Delete) and Query (Read) operations. This approach can have many advantages, such as we can manage the Read or Query operation in multiple ways; for example, at some places we need only a subset of data in list form, and at other places we might need a more detailed version.

The approach also has additional advantages as we are segregating the read operations from the update operations. We can use Read replicas of the database to achieve better performance. CQRS also helps in segregating UI views. An update to the data can trigger an event to update the view.

Though CQRS is helpful in separating Read and Update concerns, and hence supports the Single Responsibility principle, we need to be careful in implementing this as we are adding complexity to the system.

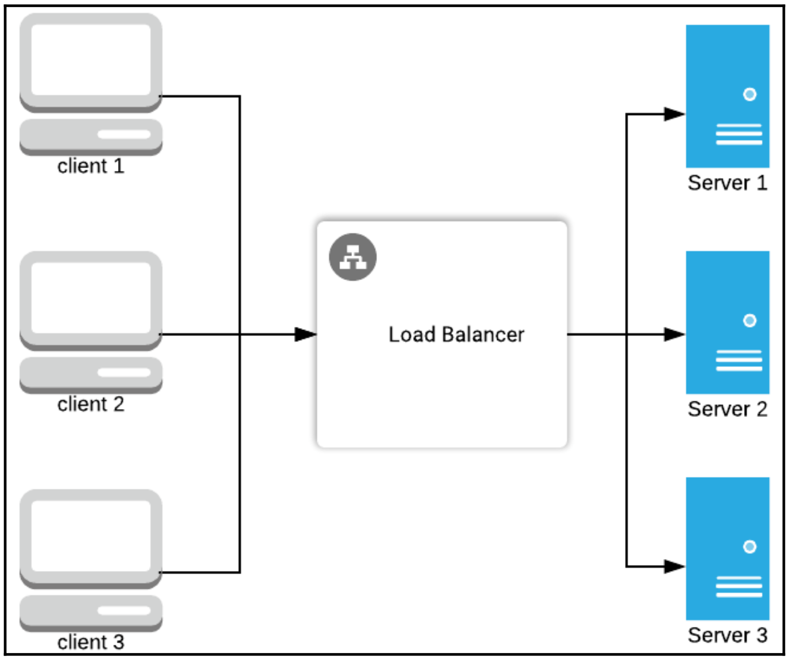
If the entity is simple and does not need too many customized operations, it might be a good idea to stick with a simple CRUD operation implementation.

Scaling Microservices with caching

We have talked about various aspects that help to scale Microservices-based applications. But our discussion is not complete without touching upon Caching. Caching is an important aspect that helps to manage load on the services, and hence helps to scale the application.

Caching can be done at different levels, most importantly on the client side and the server side. To understand how caching will help us to scale our applications easily, let's look at an example. Let's say we have a simple service that returns data for an employee ID. That is, the service takes an employee ID and returns employee data. For the sake of simplicity, let's assume this is a stateless Microservice.

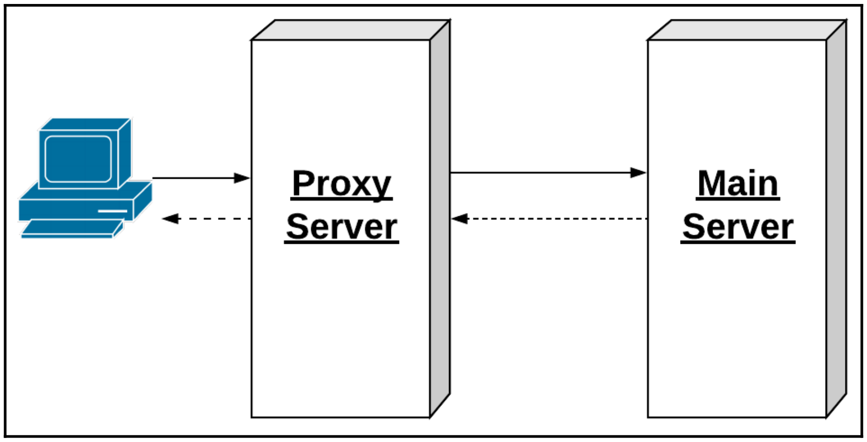
The following diagram shows the scaling of stateless Microservices:



Let's say we have a multiple clients that are making a call to our employee data service. Say we have `getEmployeeData` service, which takes an employee ID. This service is called using `API /Employee/{ID}`.

Now, imagine there are multiple clients making calls to the service. In this context, we can cache at the client level or the server level:

- **Client-side cache**: The client will maintain their own cache. For this example, let's say the client is maintaining a simple hashtable, with a key-value pair, where the key is the employee ID and the value is the employee data. Whenever the client needs data for an employee, it will first check its internal cache, and if the employee ID is found in the client-side cache, it would not need to make a call to the server, saving a round trip to the server. The results will be lightning fast in this case, as the client is serving the results internally. As with any cache, we will need to take care of certain aspects, such as an expiring cache and limiting the number of records that can be cached. The number of records that can be cached is critical in client-side caching as we are dependent on the user's machine. If the caller to the employee service is another service, we can have a greater number of records in the cache, depending on who is calling and the business requirements. Similarly, we need to make sure to expire cache records. Again, this is dependent on our business needs, such as how often we expect our employee records to be updated.
- **Server-side cache:** Caching is done at the service level rather than the caller level. The Microservice will maintain a cache on its own. There are many libraries that provide caching off the shelf, such as Jcache or Memcached. You can also use third-party caching, such as a Redis cache, or build a simple caching mechanism within the code, as per your application's need. The core idea is that we need not do all the work again while refetching some data. For example, when we ask for an employee record against an employee ID, we might be fetching data from one or more databases and doing several calculations. The first time, we will do all the tasks, but then store the record in cache. Next time, when the data is asked for against the same employee id, we will just send back the record in the cache. Again, we need to consider aspects such as expiration and cache size, as we discussed in the case of a client-side cache.
- **Proxy caching:** Proxy caching is another technique that is gaining popularity. The idea is not to directly hit the main application server. The request first goes to a **Proxy Server**, and if the required data or artifact is available on the **Proxy Server**, we can avoid a call to the main server. The **Proxy Server** is usually close to the client, mostly in the same geographical area, so it is faster to access. Moreover, it helps us reduce the load on the main server. The following diagram shows how a proxy cache works:



The proxy cache is more popular with static content, such as images, scripts, and HTML. We will need to apply expiration criteria for the contents. Mostly, a versioning system is used to manage various updates to content, such as scripts and images, to make sure that the changes get reflected immediately.

- **Distributed caching:** As the name suggests, distributed caching is a mechanism for maintaining a cache in more than one place. There are multiple ways to implement a distributed cache. In its simplest form, we just keep a copy of the cache in multiple places, which helps us to divide the load among multiple sources. This approach is useful when we are expecting too much load and the amount of data to be cached is not too great. The other example of distributed caching is when we have lots of data to cache, and we cannot create a single cache. We would divide the data to be cached into multiple caching servers. The division can be based on application requirements. In some cases, we can have the cache distributed based on geography. For example, users in India are being served from a different cache than users in the US. Another simple piece of logic for cache distribution can be based on data. For example, we cache employee details based on employee IDs on different machines, based on the first letters of their first names such as A to F on machine 1, G to M on machine 2, and N to Z on machine 3. The method can be devised based on application requirements, but the core idea is to cache data on multiple distributed machines for easy access.

Summary

In this chapter, we discussed scalability and why Microservices-based applications are a natural fit for scaling. We looked at various aspects of scalability. We talked about why stateless services are easier to scale than stateful services. Also, we discussed scaling with respect to databases, and highlighted why NoSQL databases are easier to scale. Finally, we looked at the importance of caching when scaling our application. Overall, this chapter should have given you a good idea of how to scale an application that is composed of Microservices.

In the next chapter, we will talk about securing our Microservices.

7
Securing Microservices

Due to the nature of the internet, applications are publicly served and are being accessed by a large number of users; therefore, security requirements are vital and they need to be implemented in the very early stages of the project development process – these practices should also be followed in a microservices ecosystem. In this chapter, we will show you how to secure your Microservices, by demonstrating with both Java-EE- and Spring-based implementations.

We will cover the following topics in this chapter:

- Securing Microservices with **JSON Web Token** (**JWT**)
- Java Security API – JSR 375
- Spring Security with Spring-Boot-based Microservices
- HTTPS – the secured protocol

Securing Microservices with JWT

In an architecture based on Microservices, there will definitely be a number of services that will be communicating with each other. To establish a secure medium of communication between those channels, JWT is a common and well-adopted practice to apply.

Anatomy of a JWT

JWT is an open standard (RFC 7519: `https://tools.ietf.org/html/rfc7519`) and it is represented as a JSON object that is shared safely between two parties. A JWT is a compact set of information, meaning that it's small in size. It consists of three parts: a header, a payload, and a signature. These parts get encoded and then concatenated with a dot as `header.payload.signature` to form the token. Since tokens are small in size, they can be either sent as an HTTP POST parameter or can be added as an HTTP header field itself. Small sizing will also enable a faster mode of communication.

Let's continue with the anatomy of `JWT` by detailing its aforementioned subparts: header, payload, and signature. The header defines the type and the hashing algorithm for the signature. Here is an example:

```
{
   "typ": "JWT",
   "alg": "HS256"
}
```

In the example, the type is defined as `JWT` and the hashing algorithm is set to `HMAC-SHA256`. It could alternatively be set as `RSA`, where a public/private key pair would be used to create the signature. The payload is the actual content, and it will usually be a bunch of claims that point to the information that is being transmitted. Here is an example of the payload that contains a user identification value:

```
{
   "userId": "6562ce85-7fce-4f02-a68c-17f37609d2aa"
}
```

A standard set of JWT claims are provided by default; they are not mandatory but their names are reserved, as follows:

- `aud`: The audience of the token
- `exp`: The expiration date of the token
- `iss`: The issuer of the token
- `sub`: The subject of the token
- `nbf`: Defines the time on which the JWT will start to be accepted for processing
- `iat`: The time the token was issued
- `jti`: The unique identifier for the JWT to prevent replay attacks

The pseudo code snippet given here demonstrates how a signature gets created, with the encoding first and then the hashing:

```
data = base64urlEncode(header) + “.” + base64urlEncode(payload)
signature = hash(data, "secret");
```

The hashing algorithm uses a key, which is set as `secret` in our case, to compute the signature.

It's important to understand that the data with JWT is encoded and then signed, which confirms the authenticity of the data. But this does not secure the data, since encryption didn't take place in the given example snippet.

How does JWT work for Authentication?

When a user logs into the system by providing their credentials, a `JWT` will be created and it will be returned to the user. This token should be saved locally on the browser and should be sent back to the server with each request that tries to access a protected resource. `JWT` is usually transferred in an `Authorization` HTTP header with the `Bearer` schema:

```
Authorization: Bearer <token>
```

This approach enables a stateless authentication mechanism where the user state is never stored on the server's memory. The bearer tokens should be protected in storage and transport in order to prevent any misuse. An important advantage we get with this approach is implementing scalability. We have already covered the JWT in the context of scalable Microservices in `Chapter 6`, *Scaling Microservices*, with examples.

Java Security API – JSR 375

Until recently, there were no standards for implementing Java Security. Don't get me wrong, there were many libraries and frameworks available, but a formal standard specification was missing. **Java Specification Request** (**JSR**) 375 deals with the problems caused by a lack of standards. In this section, we will discuss JSR 375 in detail and we will also take a look at its reference implementation Soteria. To get started with Soteria, all you need to do is add the following dependencies in your Maven:

```
<dependency>
<groupId>org.glassfish.soteria</groupId><artifactId>javax.security.enterpri
se</artifactId><version>1.0</version>
</dependency>
```

Check out `https://mvnrepository.com/artifact/org.glassfish.soteria/javax.security.enterprise` for the latest Soteria implementations.

JSR 375 is focused on the following core aspects:

- Providing an API for authentication through the interface `HTTPAuthenticationMechanism`
- An identity store API through `IdentityStore`
- A security context API through `SecurityContext`

We will cover these three APIs in subsequent subsections.

The HTTPAuthenticationMechanism API

As a part of the Java Security API, the container must provide `HttpAuthenticationMechanism` implementations for three authentication mechanisms. The three implementations are:

- `Basic HTTP authentication`
- `Form-based authentication`
- `Custom-form authentication`

Each of these implementations can be triggered by the use of the following annotations:

- `@BasicAuthenticationMechanismDefinition`
- `@FormAuthenticationMechanismDefinition`
- `@CustomFormAuthenticationMechanismDefinition`

When we use any of these annotations in the code, the container will instantiate an instance of the associated implementation.

Let's take a look at these three mechanisms, one by one.

Basic HTTP Authentication

Usage of `@BasicAuthenticationMechanismDefinition` informs the container that code would the be using `BasicHTTPAuthentication` mechanism. This has one optional parameter, `realmName`, which contains the name of the realm that is sent along with `WWW-Authentication Header`, as follows:

```
@BasicAuthenticationMechanismDefinition(
 realmName = "userRealm")
 @ApplicationScoped
 public class AppConfig{
....
}
```

When the code given here is used, if the container receives an unauthenticated request, it will inform the client to send the authentication details with `WWW-Authentication header`. The client then needs to provide authentication details for the successful execution of the request.

Form-based Authentication

`@FormAuthenticationMechanismDefinition` lets the container know that the web application is using form-based authentication. The annotation has one mandatory parameter, `loginToContinue`, which in turn takes the annotation details of `@LoginToContinue`. `@LoginToContinue` can provide details of login and error pages. The client would be sent to a login or error page based on the state of authentication, as follows:

```
@FormAuthenticationMechanismDefinition(
 loginToContinue = @LoginToContinue(
 loginPage = "/login.html",
 errorPage = "/login-error.html"))
 @ApplicationScoped
 public class AppConfig{
   ...
 }
```

Custom form-based Authentication

Like form-based authentication, `@CustomFormAuthenticationMechanismDefinition` provides flexibility to add a custom way of authentication using J2EE technologies, such as JSF, as follows:

```
@CustomFormAuthenticationMechanismDefinition(
loginToContinue = @LoginToContinue(loginPage = "/login.do"))
@ApplicationScoped
public class AppConfig
{
   ....
}
```

We can see that this is similar to the form-based authentication, the only difference is we are specifying a custom server link that is responsible for providing form details and the underlying authentication mechanism.

Identity Store

Identity Store is basically a data store that keeps information such as usernames, group memberships, and credentials. The Java Security API provides IdentityStore abstraction in the form of IdentityStore. IdentityStore can work well with HTTPAuthenticationMechanism, but you are free to use any other authentication mechanism. The IdentityStoreHandler API provided with IdentityStore manages instances of IdenityStore. The Java Security API comes with a default implementation of IdentityStoreHandler. The default implementation can authenticate against multiple instances of IdentityStore and is sufficient in most cases, though one can implement a custom IdentityStoreHandler. IdentityStoreHandler would iterate over stores and return CredentialValidationResult, which in its simplest form would return a status value as one of three states: NOT_VALIDATED, INVALID, or VALID. If a VALID state is returned by IdentityStore, no more stores are checked and the result is returned. If INVALID is returned by IdentityStore, further stores are checked. If all the states are INVALID, it is considered the final state, otherwise NOT_VALIDATED is returned.

Built-in and Custom IdentityStores

There are two major built-in identity stores available, for RDBMS and LDAP.
The @DataBaseIdentityStoreDefinition annotation is used to configure an RDBMS-based Identity Store, as follows:

```
@DatabaseIdentityStoreDefinition(dataSourceLookup =
"jndiPathForDataStore",callerQuery = "Query to fetch user
details",groupsQuery = "Query to fetch group details",priority=30)
```

Similar to database-based identity stores, we can use a built-in annotation, @LdapIdentityStoreDefinition, for an LDAP-based Identity Store:

```
@LdapIdentityStoreDefinition(url = "ldap://localhost:10389",callerBaseDn =
"ou=caller,dc=example,dc=com",groupSearchBase =
"ou=group,dc=example,dc=com")
```

In most cases, built-in stores are sufficient for `IdentityStore`, but if required, one can create a custom implementation. The `IdentityStore` interface provides four methods, all of which have default implementations. One can override one or all of the methods based on requirements. Here are the four methods:

- `default CredentialValidationResult validate(Credential credential)`: This is a validate method, which as the name suggests, is responsible for validating the given credentials and returns `CredentialValidationResult`.
- `default Set<String> getCallerGroups(CredentialValidationResult validationResult)`:The method `getCallerGroups`, is responsible for returning set a of groups which the user is associated with.
- `default int priority()`: The priority method, only comes into play if more than one `IdentityStore` is available; the one with the lowest value will be given the highest priority.
- `default Set<ValidationType> validationTypes()`: Here, `validationTypes` returns a set of `ValidationType` being implemented by the current setup.

The security context API

The security context provides access to security-related information about the current user. `SecurityContext` consists of five methods, as follows:

- `getCallerPrincipal`: If the current user is authenticated, this method returns the container-specific principal to the user.
- `getPrincipalsByType`: Returns all the principals of the given type.
- `isCallerInRole`: Returns true if the current user is part of the role sent as a parameter.
- `hasAccessToWebResource`: Determines whether the current user has access to the web resource passed as a parameter.
- `authenticate`: Triggers the authentication method set up for the application.

Spring Security with Spring-Boot-based Microservices

Spring Security is an authentication and authorization framework for Java-EE-based applications. It provides a comprehensive set of features that handle your security requirements and also offers extension points in order to apply application-specific customizations easily. We will use Spring Boot to create Spring-based applications, and we will use the starter kit for Spring Security that Spring Boot provides.

At the time of writing, the current version of Spring Boot is `2.0.0.RELEASE`.

Spring Boot simplifies the process of applying security measures to the application by simply adding its dependency to the `Maven POM` file:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
   <version>2.0.0.RELEASE</version>
</dependency>
```

Version `2.0.0.RELEASE` of `spring-boot-starter-security` defines Spring security version `5.0.3.RELEASE` as a transitive dependency under the hood.

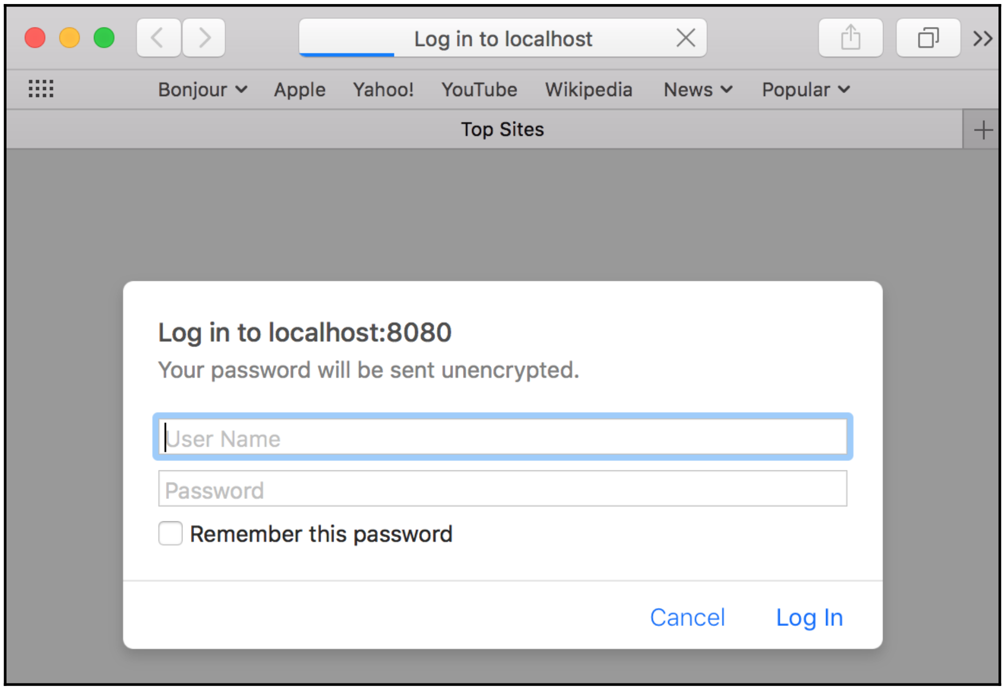
When the `spring-boot-starter-security` artifact is added to the classpath, the entire application is protected with HTTP Basic Authentication.

HTTP Basic Authentication is one of the most widely used authentication mechanisms; it secures web resources by requesting the username and password sent by a client. It uses standard fields in the HTTP header and doesn't require any cookies or session identifiers. With HTTP Basic Authentication, the username and password are passed as *Base64*-encoded, and due to *Base64* being a reversible encoding mechanism, it's important to mention that the *HTTP Basic Authentication* scheme is not secure and HTTPS/TLS should be used in conjunction with it to secure the transport layer.

To secure a Spring-Boot-based REST service implementation, we are going to use the same implementation that we had in `Chapter 2`, *Creating Your First Microservice* and we will just add the `spring-boot-starter-security` dependency. The `WeatherServiceApplication` class contains the following executable `main()` method. The example contains the temperature Microservice implemented in `Chapter 2`, *Creating Your First Microservice*.

```
@SpringBootApplication
public class WeatherServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(WeatherServiceApplication.class);
    }
}
```

After executing the method on `WeatherServiceApplication` and requesting the `http://localhost:8080/temperature` REST endpoint, we'll come across the following login page:



Log in to localhost:8080

Your password will be sent unencrypted.

User Name

Password

Remember this password

Cancel Log In

By accessing a secured endpoint, we will be prompted with a login page that asks us to input our credentials. The default **User Name** for this login will be `user`. The **Password** will be printed on the console while executing the `main()` method as:

```
Using default security password: 7b559a60-1673-4308-8dd4-c441732b0c70
```

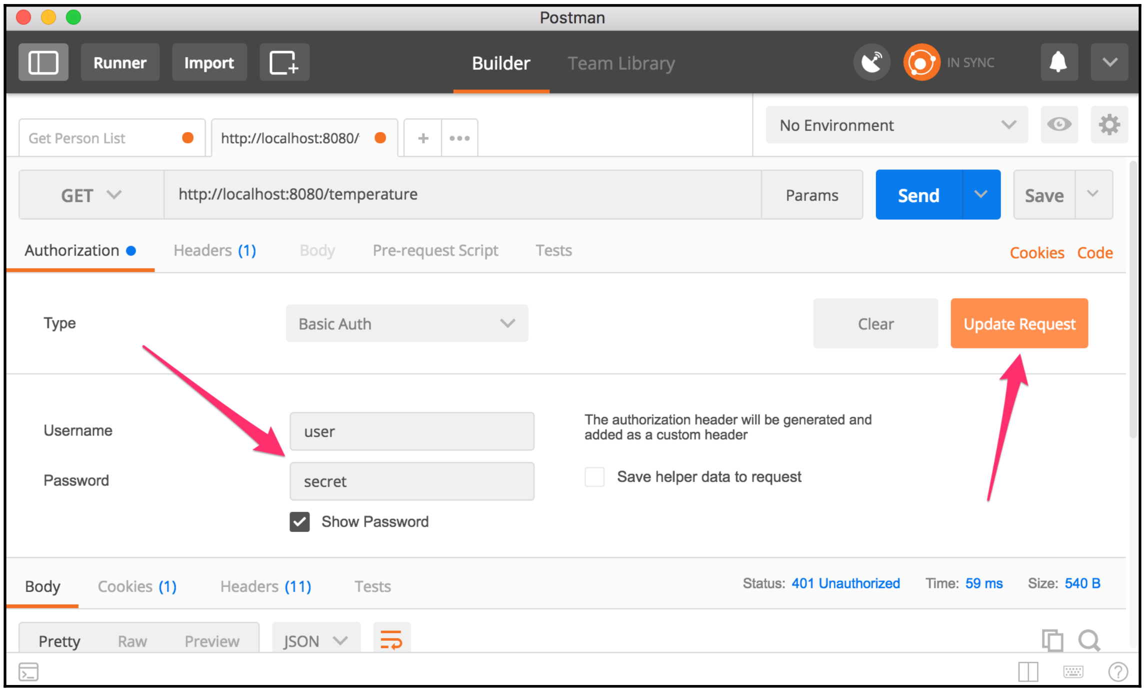
It's a password that was generated randomly while the application was starting so it should change at each execution of the method. It's also possible to request the endpoint with `curl` by providing the credentials, as follows:

```
curl -u user:7b559a60-1673-4308-8dd4-c441732b0c70
http://localhost:8080/temperature
```

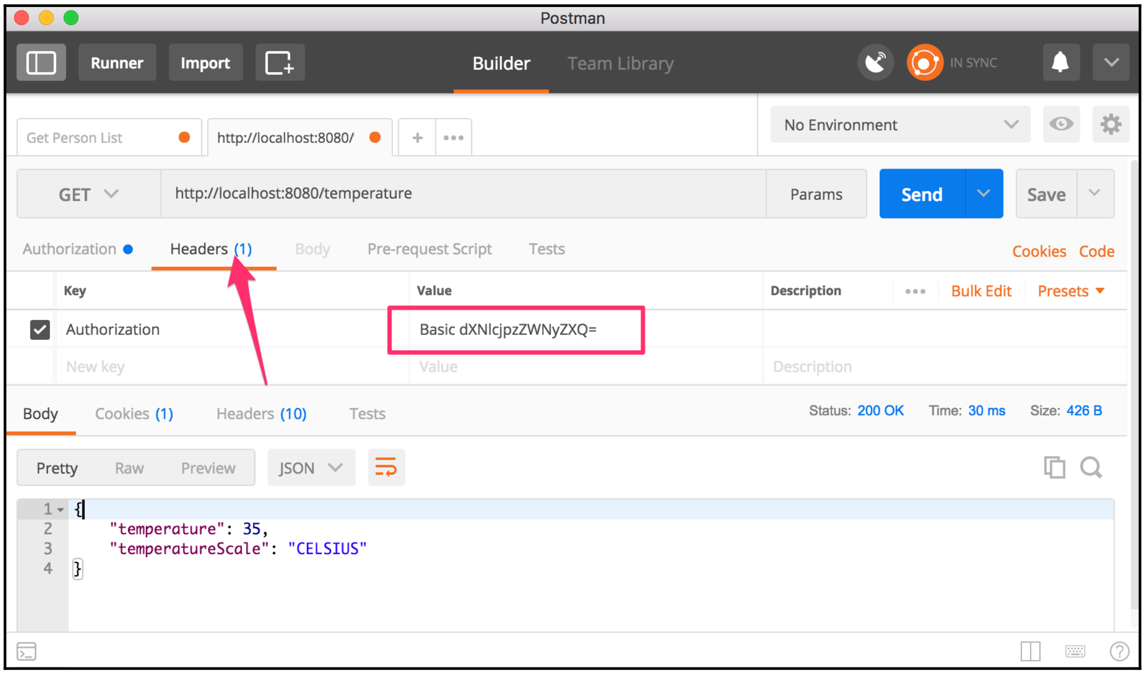
Having dynamically generated a password for your application will not necessarily get it into a production-ready state, so you set the password could by providing the username and password through a configuration file. Spring Boot has a key-value file template, `application.properties`, to externalize the configuration and it should reside under the root folder of the classpath by default. In our example, its content will define the default username and its corresponding password as follows:

```
spring.security.user.name=user
spring.security.user.password=secret
```

After creating this properties file under the `src/main/resources` folder of our application, we can try to request the endpoint with Postman (`https://www.getpostman.com`) this time. To provide the credentials, we need to set the authentication type as **Basic Auth** and provide the **Username** and **Password** defined in the properties file. The final task should be to update the request before clicking the **Update Request** button:



As you can see on the **Headers** tab in the following image, the **Authorization** header is set to Basic type along with a Base64-encoded value:



GET http://localhost:8080/temperature

Key	Value	Description
Authorization	Basic dXNlcjpzZWNyZXQ=	
New key	Value	Description

Status: 200 OK Time: 30 ms Size: 426 B

```
{
    "temperature": 35,
    "temperatureScale": "CELSIUS"
}
```

If we decode that encoded value, we will see that the Username and Password are concatenated with `:`, as follows:

```
user:secret
```

In a real-world scenario, you will probably want to match your user's credentials across to a database or an LDAP realm, instead of a configuration file. To tighten your security measures, we will look at an example for an In-memory user realm, and then enhance it with a database-based realm.

Configuring Spring Security with the In-memory realm

In order to have the user's credentials defined In-memory, first we need to provide a custom Spring Security configuration:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()
                .passwordEncoder(passwordEncoder())
                .withUser("user1")
                .password(passwordEncoder().encode("secret1"))
                .roles("USER");
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().fullyAuthenticated();
        http.httpBasic();
        http.csrf().disable();
    }
}
```

The SecurityConfig class extends WebSecurityConfigurerAdapter, which is a base class for creating the WebSecurityConfigurer instance, and it will allow us to perform the customization by overriding its methods. The first configure() method allows us to configure an authentication manager by passing AuthenticationManagerBuilder as a parameter. That will help us build an in-memory authentication, LDAP authentication, or a JDBC-based authentication. We are also providing a custom password encoder that is created by the passwordEncoder() method. With Spring Security 5, we need to set the password encoder, thus BCryptPasswordEncoder is used for demonstration purposes. The second configure() method configures an instance of HttpSecurity by providing an HTTP authentication mechanism. The @EnableWebSecurity annotation is a marker annotation added onto the SecurityConfig configuration class to execute the customized configuration methods.

So, we will have user1 defined in our In-memory realm and we can send requests to our protected Temperature Microservice, as follows:

```
curl -u user1:secret1 http://localhost:8080/temperature
```

In the next section, we will integrate a database realm in order to store the credentials of the user in a persistence mechanism.

Configuring Spring Security with the database realm

To make our application product-ready, we need to enhance the storage of the configurations. One of the most common ways to store user credentials is to use a database to handle user information, roles, and their mappings between each other. In order to have our database configuration ready, we first need to add the spring-boot-starter-data-jpa dependency, which provides a set of convenient dependency descriptors that help us use JPA for database connectivity:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
   <version>2.0.0.RELEASE</version>
</dependency>
```

spring-boot-starter-data-jpa uses the Hibernate object/relational mapping tool (http://hibernate.org/orm) under the hood.

We will use the H2 database in our example for its simplicity and its In-memory database support. You can use your favorite database instead by simply changing the Datasource configuration. Here is the Maven dependency for H2:

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <version>1.4.197</version>
</dependency>
```

First, we need to define the JPA entities for the `User` and `Role` classes, as follows:

```
@Entity
@Table(name = "app_user")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "id")
   private Long id;

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

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

   @Column(name = "name")
   private String name;
   @ManyToMany(fetch = FetchType.EAGER)
   @JoinTable(name = "user_role", joinColumns =
      @JoinColumn(name = "user_id",
         referencedColumnName = "id"),
         inverseJoinColumns =
            @JoinColumn(name = "role_id",
               referencedColumnName = "id"))
    private List<Role> roles;

    // getters & setters
}

@Entity
@Table(name="app_role")
public class Role {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column(name="role_name")
```

```
        private String roleName;

        // getters & setters
    }
```

HTTPS – The Secured Protocol

Before closing this chapter, we would like to touch upon one of the simplest tools we have to secure communication between client and server, or between different Microservices while communicating through REST. The tool we are referring to here is the secured HTTPS protocol. It is almost always recommended to use HTTPS for production-level REST communications where data is of importance. Let's take a look at what HTTPS is and how it helps.

HTTP stands for **Hypertext Transfer Protocol**, over which most communication happens on the internet. HTTPS can be thought of as HTTP + Security, or HTTP over **SSL** (**Secure Socket Layer**) or **TLS** (**Transport Layer Security**, the successor of SSL).

The oldest and most trustworthy form of data security is encryption, which can be implemented in two ways: a symmetric manner or an asymmetric manner. Asymmetric encryption is implemented using a single key, for both the encryption and decryption processes. To implement asymmetric encryption, we use a set of keys, namely a public and a private key. The data encrypted through one key can be decrypted through the other. Asymmetric communication is slightly slower due to the additional complexity.

Another important aspect we need to understand is certificates. There are various certification authorities that provide certificates for domains. Whenever you visit an HTTPS link, you see a small lock icon on the left side of the address bar; if you click on this icon, you will see certificate details, which include the authenticity of the website and the authority that is providing this certificate. If you click a spam link by accident, validating the certificate might help you save yourself.

So how can https make sure two services communicate in a secured manner? Let's take a step-by-step look at what happens behind the scenes when we use HTTPS-supported URLs:

1. When a call is made through the HTTPS URL, the server treats this as a special request that needs security.
2. The server returns a certificate to the calling service or browser, along with a public key. Note that the server will never share a private key.

3. The calling service or browser authenticates the certificate and makes sure the communication is being done with the correct server.
4. The calling service or browser generates a symmetric key, encrypts it using the public key it received from the server, and sends the encrypted key to the server.
5. The server receives the encrypted symmetric key and decrypts it using its private key.
6. Both the client and server have a symmetric key that was transferred securely. Now both parties use this symmetric key for the rest of the communication.

We can see the usage of HTTPS adds a very strong layer of security for communications happening over the internet.

Summary

In this chapter, we covered the basics of security requirements while we are building an application with a Microservices-based design. We started with a JWT-based security implementation, and then discussed JSR 375 (the Java Security API). Following that, we talked about Spring Security and its implementation with respect to Spring-Boot-based Microservices. Finally, we discussed the usage of HTTPS when communicating over the internet.

In the next chapter, we will focus on monitoring Microservices-based applications.

8
Monitoring Microservices

So far, we have discussed the creation, scalability, and security of Microservices. In this chapter, we will focus on the monitoring of Microservices.

Monitoring Microservices is important, as you would definitely like to keep tabs on the health of services to make sure all requests are served without any errors and in a timely manner. Monitoring an application that uses Microservices is different from monitoring a monolithic application where everything is deployed on a single instance.

In this chapter, we will discuss monitoring Microservices, the challenges that can be expected, and how to overcome these challenges. We will be using certain tools and see examples to showcase the implementation of monitoring in an application built using Microservices, but, the user is free to use any tools or methods they like. The idea of this chapter is to explain the importance of monitoring Microservices, the core principles, and the challenges involved.

The following topics will be covered in this chapter:

- What is monitoring and why is it required?
- Understanding the core concepts and terms
- Taking a closer look using an example
- Tools for monitoring Microservices

What is monitoring and why is it required?

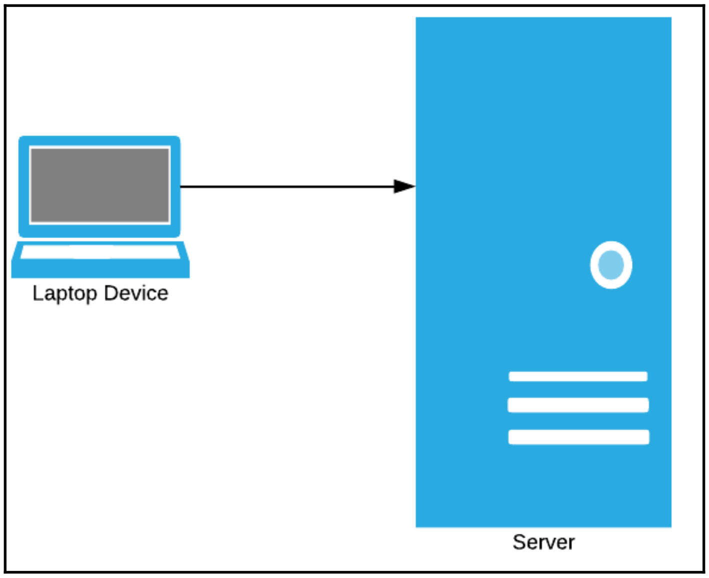
Whenever we create an application, it goes through multiple phases, such as requirement analysis, design, development, and finally deployment. The real test of an application starts after deployment. Think of a scenario where a user of an application comes to a development team complaining that they are not able to access the application, or some of the pages are throwing an error or behaving in an unexpected manner.

Where should we start looking for the problem? Is there a problem with the current user account or with the application itself? Is some code not behaving as it should?

Monitoring the application comes to the rescue. It not only helps us to find issues in cases such as those just mentioned, but also helps us to proactively understand the problems in our application, such as whether a service is down or taking too much time to respond.

Monitoring Microservices

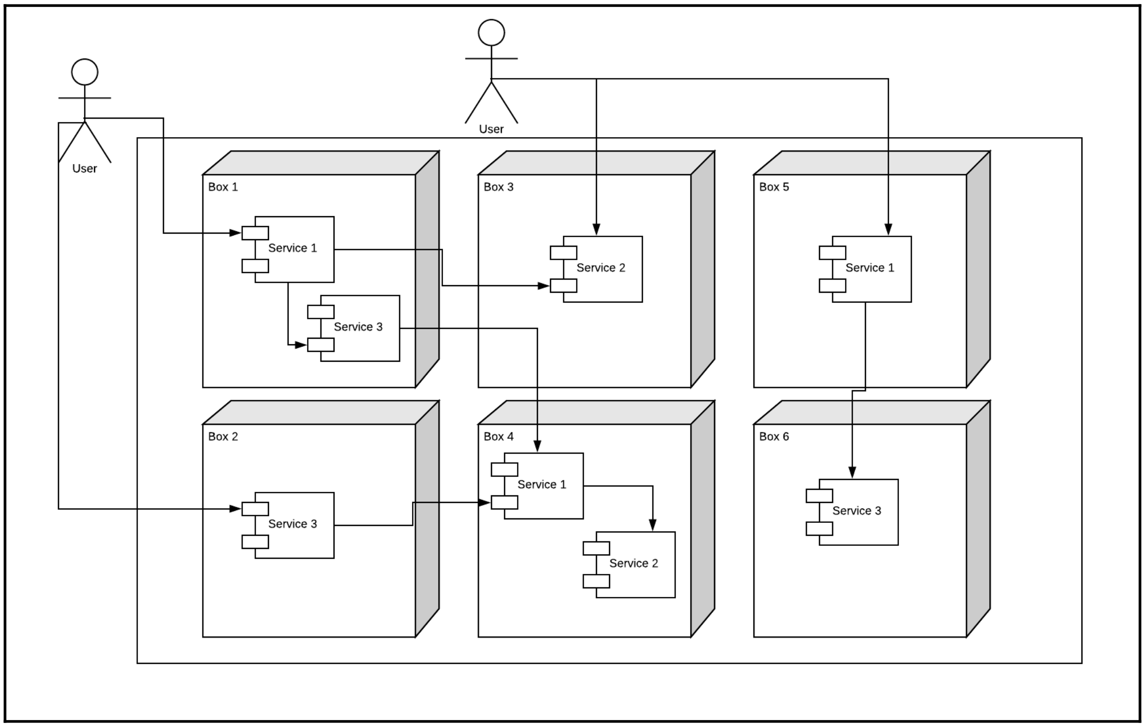
Let's look at a scenario where we are dealing with a very simple monolithic application deployed on a single server:



We can see all the user requests through the browser or other clients will directly hit our application deployed on the server. So when an issue or latency is reported, we know where to look for the problem: you log in to the server, and look at logs and server health. We know where to add all the monitoring and profiling. Well, in the real world, a deployment will be more complex as it will have databases, log files, clusters, load balancers, and so on, but, it is still possible to monitor applications with a monolithic architecture.

Now, let's think about a simple Microservices-based application. An application that implements Microservices will often have multiple Microservices, each one focused on one core solution and interacting with other services to pass on or fetch information.

Take a look at the following diagram to get an idea of the complexity that Microservices can add to the system:



In the preceding diagram, we are trying to show instances of three services deployed on six boxes. Again, to keep things simple, we won't get into Database, Queue, or scalability details. Think of a scenario where you get a call from a user saying that they are not able to access a page or are facing some latency. We don't even know where to start, which box to look at, or which service is down or slow to respond.

Consider the fact that the preceding design is simple compared to a real-world scenario where there might be tens of services with auto-scaling, queues, and messaging, and where services will be spawning up or down as required. Think of the mess we will be dealing with.

These complexities make it very important that we add, monitor, and profile carefully when dealing with Microservices. Now, let's deal with the actual implementation of monitoring in Microservices.

Understanding core concepts and terms

Before getting into an example to see how monitoring is done, let's discuss a few core concepts and terms that will help us understand the monitoring and profiling of Microservices:

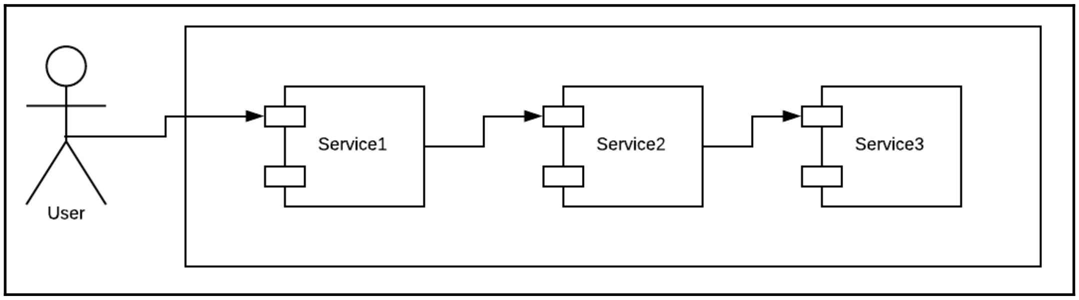
- **Monitoring** is an umbrella term that is used to talk about various aspects, such as general health checks of services, latency, logging, resource usage, and checking the well-being of the services and applications.
- **Profiling** is about observing delays and understanding how much time each service is taking. It will help us to understand which services are taking time and pinpoint the problem areas.
- **Tracing** is more about tracking the flow of control when a request is fired, more or less similar to profiling but with some different details. Distributed tracing extends the concept to a distributed system that has multiple Microservices deployed independently. For example, when the user hits a URL, which API is getting executed, which might call another service, and so on. We would like to know how the flow is moving and about the health of each service. Is there a service that is not responding or is slow? Distributed tracing will help us with this.
- **Logging** can be anything; we just log all the events with parameter details. We can also log critical areas of the service or application, which can later help us understand what happened behind the scenes. Log monitoring can be done manually, or be automated through the use of log monitoring tools.
- **Metrics** are another important way to look at the health of your system. These can be produced using logs or tracing data that show how much time various services are taking, to see whether something needs special attention. Different types of metrics can be generated on an as per-need basis and provide information about the general health of the system at a glance.

- **Health checks** are automated scripts or tools that keep tabs on the health of the services. This includes the health of hardware infrastructure as well as the availability of different services.
- **Alerting** is the system that helps trigger an action when an unwanted or error condition is observed in the system. Email or messaging alerts can be sent based on need, and an escalation policy can be set up as per the system's requirements.

If you are using a third-party cloud service provider, such as **Amazon Web Services** (**AWS**), Google Cloud, or Microsoft Azure, you might be getting some of the services as part of these package to monitor Microservices.

Taking a closer look using an example

The best way to understand monitoring is through an example. We will take a very simple scenario where we have three services; these services will be calling each other sequentially:

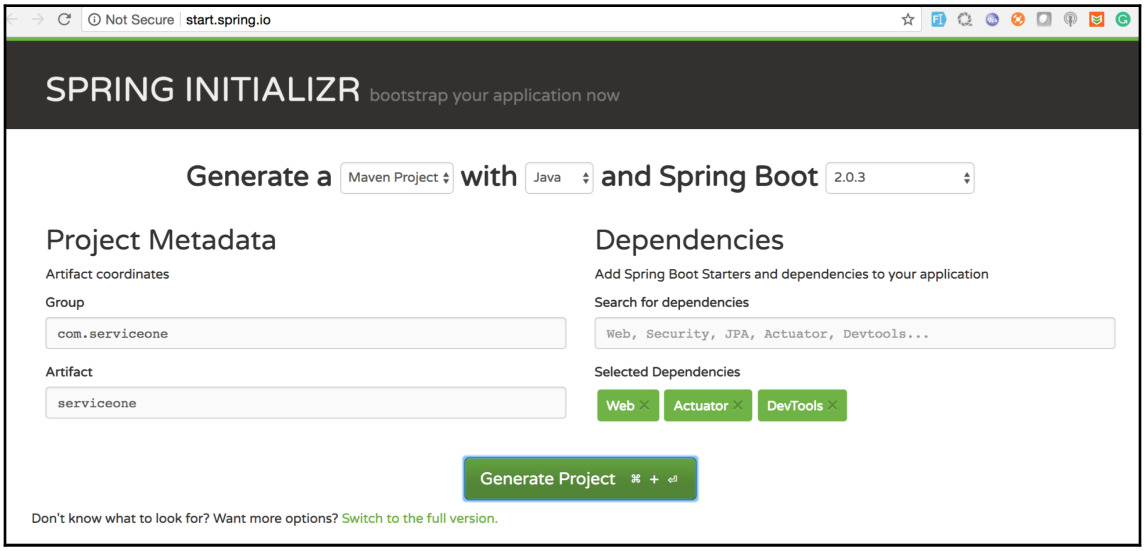


We will try to look at various problem scenarios and monitor them with a tool called Zipkin.

Creating the example services

Let's create our first service. We will create the service using Spring initializer (`http://start.spring.io/`).

The following screenshot shows the **start.spring.io** interface where we are creating the service:



Not Secure | start.spring.io

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 2.0.3

Project Metadata

Artifact coordinates

Group: com.serviceone

Artifact: serviceone

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: Web, Actuator, DevTools

Don't know what to look for? Want more options? Switch to the full version.

We can import this service to the IDE as a Maven project. We can see a Spring Boot application is created.

We have used Actuator and Devtools dependencies for generating. Here are some sample dependencies you might see in `pom.xml`:

```
<dependencies>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

 <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-devtools</artifactId>
```

```
      <scope>runtime</scope>
    </dependency>

  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

The preceding code gives us an autogenerated class, as follows:

```
@SpringBootApplication
public class ServiceoneApplication
{
 public static void main(String[] args)
 {
  SpringApplication.run(ServiceoneApplication.class, args);
 }
}
```

Next, we create a simple controller:

```
@RestController

public class HelloController
{
 @GetMapping("/sayhello")
 public String sayHello()
 {
  return("Hello");
 }
}
```

We can run the application and access the service at `http://localhost:8080/sayhello`.

Now, we will add a couple more services. We will use the same approach as before to create a Spring Boot application. We will create two additional projects to implement the `hello2` and `hello3` Microservices.

Starting from `service 3`, we will make a simple service returning a string for the sake of this example, as follows:

```
@RestController
public class HelloController
{
 @GetMapping("/sayhello3")
 public String sayHello()
```

```
     {
      return("Hello from service 3.");
     }
    }
```

`service 2` will call `service 3` and add its own input along with that, as follows:

```
@RestController
public class HelloController
{
 private final RestTemplate restTemplate;
 public HelloController(RestTemplate restTemplate)
 {
  this.restTemplate = restTemplate;
 }
 @GetMapping("/sayhello2")
 public String sayHello()
 {
  String responseFromService =
restTemplate.getForObject("http://localhost:8082/sayhello3",
String.class);
  return("Hello from service 2. " + responseFromService);
 }
}
```

Similarly, our main service will call `service 2`:

```
@RestController
public class HelloController
{
 private final RestTemplate restTemplate;
 public HelloController(RestTemplate restTemplate)
 {
  this.restTemplate = restTemplate;
 }
 @GetMapping("/sayhello")
 public String sayHello()
 {
  String responseFromService =
restTemplate.getForObject("http://localhost:8081/sayhello2",
String.class);
  return("Hello from service 1. " + responseFromService);
 }
}
```

Now, let's try calling our service:

```
curl http://localhost:8080/sayhello
```

This will return the `Hello from service 1. Hello from service 2. Hello from service 3` string.

If one of the services is responding slowly, or has stopped responding, we will get a delayed response or no response at all. For example, let's stop `service 3` and try calling the main service.

We receive `{"timestamp":"2018-07-22T12:47:54.792+0000","status":500,"error":"Internal Server Error","message":"500 null","path":"/sayhello"}` as a response.

Looking at this response, we cannot make any sense out of it. This only says that the service responded with an error, but if the main service or any other service,is failing we cannot make any sense out of the response.

Similarly, if one of the services is responding slowly, we will not know what is causing the delay. Let's modify the service:

```
@GetMapping("/sayhello3")
 public String sayHello()
{
  try
 {
  Thread.sleep(2000);
 }
 catch (InterruptedException e)
 {
  e.printStackTrace();
 }
  return("Hello from service 3.");
}
```

We can see the main service has been responding with a delay of two seconds, but we do not know why.

We have seen the problems that can occur with Microservices and we will be clueless without proper monitoring implemented. There are multiple ways to implement monitoring and tracing. We can add manual code snippets as logs and monitor logs later, we can use **aspect-oriented programming** (**AOP**) to generate logs, or we can use any of the tools available to implement tracing and monitoring.

There are a lot of tools available to implement the monitoring of services; for this chapter, we will focus more on the approach and the reader can use the tool of their choice. We will try to understand how to implement monitoring using certain tools. Let's take a look at Zipkin to implement monitoring and tracing.

Monitoring Microservices with Zipkin

One of the most important aspects of monitoring while dealing with Microservices is the implementation of tracing or distributed tracing. We need to understand the availability of various services, check whether a service is slow, and take action accordingly. To showcase the use and importance of distributed tracing, we will implement it in our previous example using Zipkin. Zipkin is a popular and easy-to-use tool that help us implement tracing in Microservices.

> *"Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in Microservice architectures. It manages both the collection and lookup of this data. Zipkin's design is based on the Google Dapper paper."* - `https://zipkin.io/`

This definition mentions Google's Dapper paper. This paper talks about the implementation of tracing in distributed systems that consist of multiple Microservices. This concept forms the basis of many modern tracing systems, such as Zipkin and OpenTracing. A complete discussion on Dapper is outside of scope of this chapter, but the following example should give an idea. For readers interested in the paper, check it out at `https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf`.

Coming back to Zipkin, you can download and install the Zipkin tool using the following command:

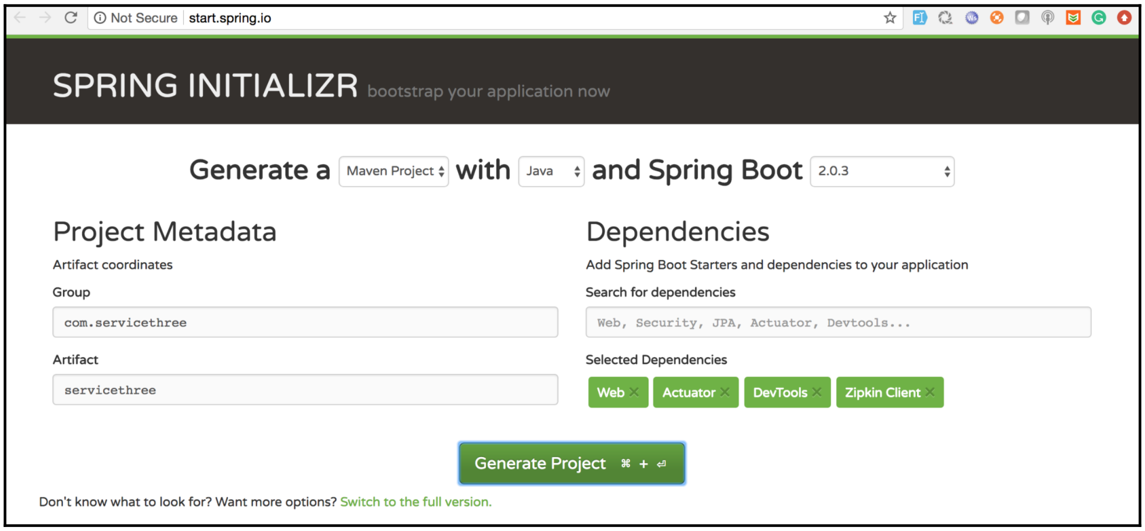
```
curl -sSL https://zipkin.io/quickstart.sh | bash -s
```

To start Zipkin independently, run the following:

```
java -jar zipkin.jar
```

Getting to the code for Microservices, we will reuse the code from the previous example. We just need to add `Zipclient` to our dependencies. We can either do this while creating the project on `http://start.spring.io/` or add the dependencies to `pom.xml`.

The following screenshot shows the **start.spring.io** interface, this time with Zipkin Client:



Not Secure start.spring.io

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 2.0.3

Project Metadata

Artifact coordinates

Group

com.servicethree

Artifact

servicethree

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

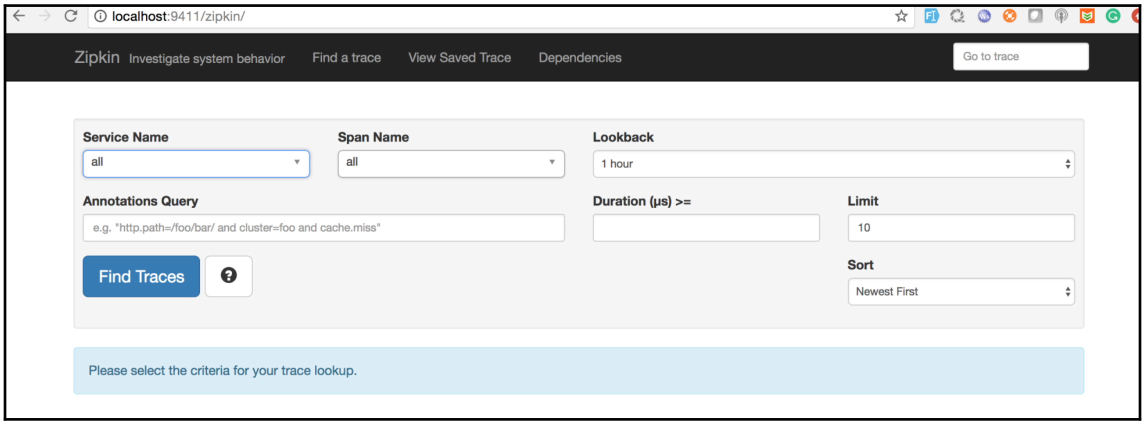
Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Web Actuator DevTools Zipkin Client

Don't know what to look for? Want more options? Switch to the full version.

The **Zipkin** interface lets you filter out various options:



localhost:9411/zipkin/

Service Name: all

Span Name: all

Lookback: 1 hour

Annotations Query: e.g. "http.path=/foo/bar/ and cluster=foo and cache.miss"

Duration (μs) >=

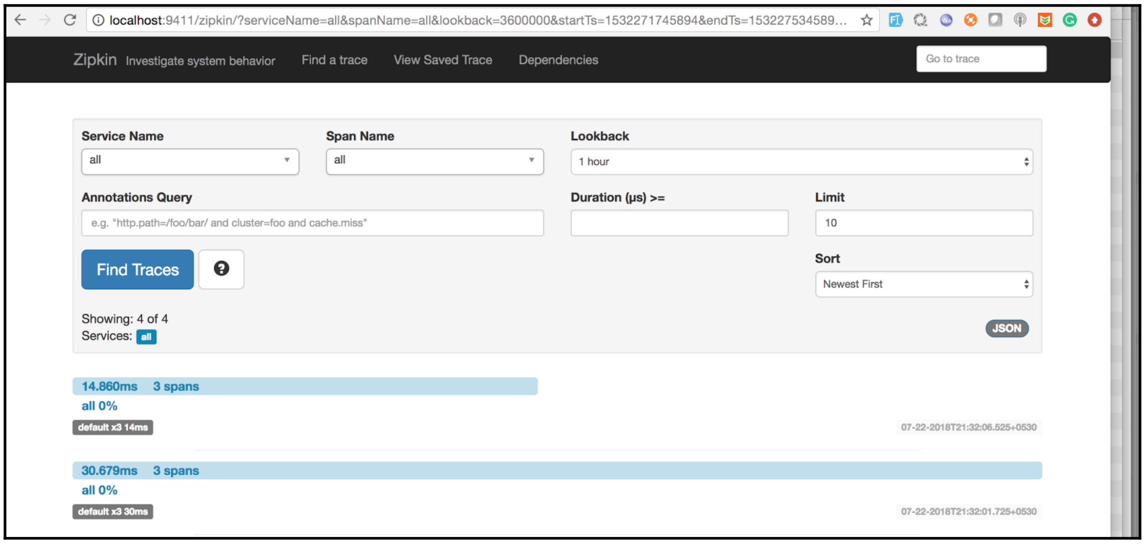
Limit: 10

Sort: Newest First

Please select the criteria for your trace lookup.

Normally, we use Sort based on **Newest First** to view the latest services at the top.

The following screenshot shows various traces, with the latest listed first:


localhost:9411/zipkin/?serviceName=all&spanName=all&lookback=3600000&startTs=1532271745894&endTs=153227534589…
Zipkin Investigate system behavior
Find a trace
View Saved Trace
Dependencies
Go to trace

Service Name: all
Span Name: all
Lookback: 1 hour
Annotations Query: e.g. "http.path=/foo/bar/ and cluster=foo and cache.miss"
Duration (μs) >=
Limit: 10
Sort: Newest First

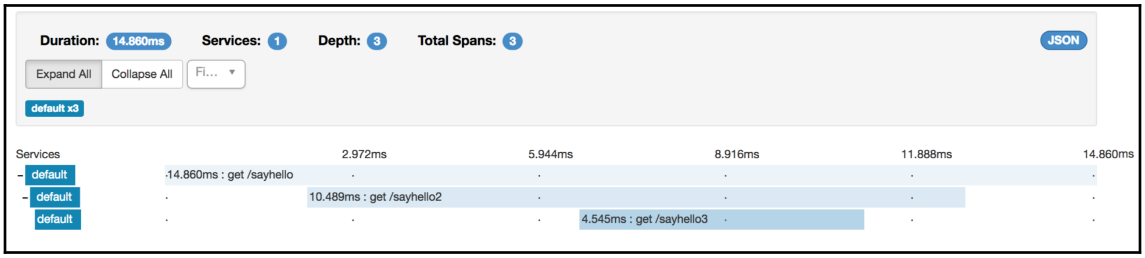
Showing: 4 of 4
Services: all

14.860ms 3 spans
all 0%
default x3 14ms 07-22-2018T21:32:06.525+0530

30.679ms 3 spans
all 0%
default x3 30ms 07-22-2018T21:32:01.725+0530

We are looking at a happy case here with services returning normally. This also shows that sometimes, the same flow might be taking more time than others. We can click on a row to further explore the flow and the time taken by different services.

The following screenshot shows a flow and how much time each Microservice is taking:



Duration: 14.860ms Services: 1 Depth: 3 Total Spans: 3

```
Services          2.972ms      5.944ms      8.916ms      11.888ms      14.860ms
- default    14.860ms : get /sayhello
 - default         10.489ms : get /sayhello2
    default                       4.545ms : get /sayhello3
```

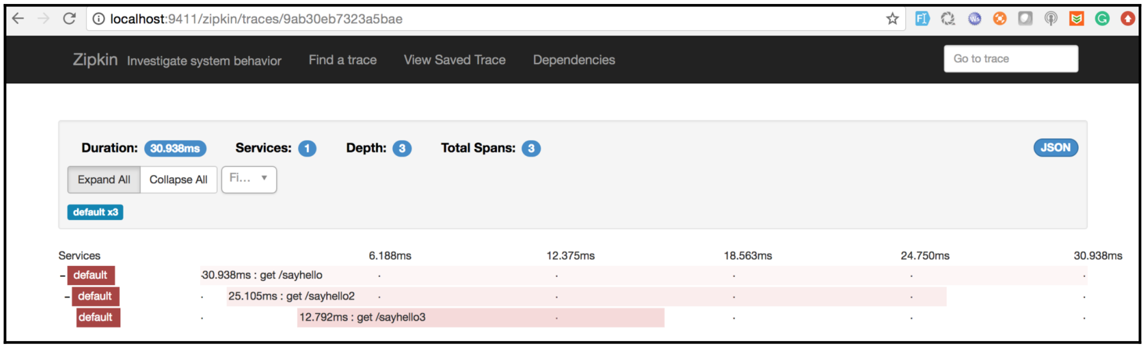
We can see the complete details for the call we made; it shows the internal calls and how much time each call is taking.

To see the advantages of Zipkin in terms of analyzing problems, let's take a look at two different cases.

Case 1 – service is unresponsive

To simulate a real-world scenario, let's say one of our services is down or is corrupted due to a recent update. In such a case, our main sayhello service will return an error, likely a five hundred internal server error, but it will not tell us what's happening behind the scenes. We are not sure which service has a problem. Let's see whether taking a look at the Zipkin interface can helps us.

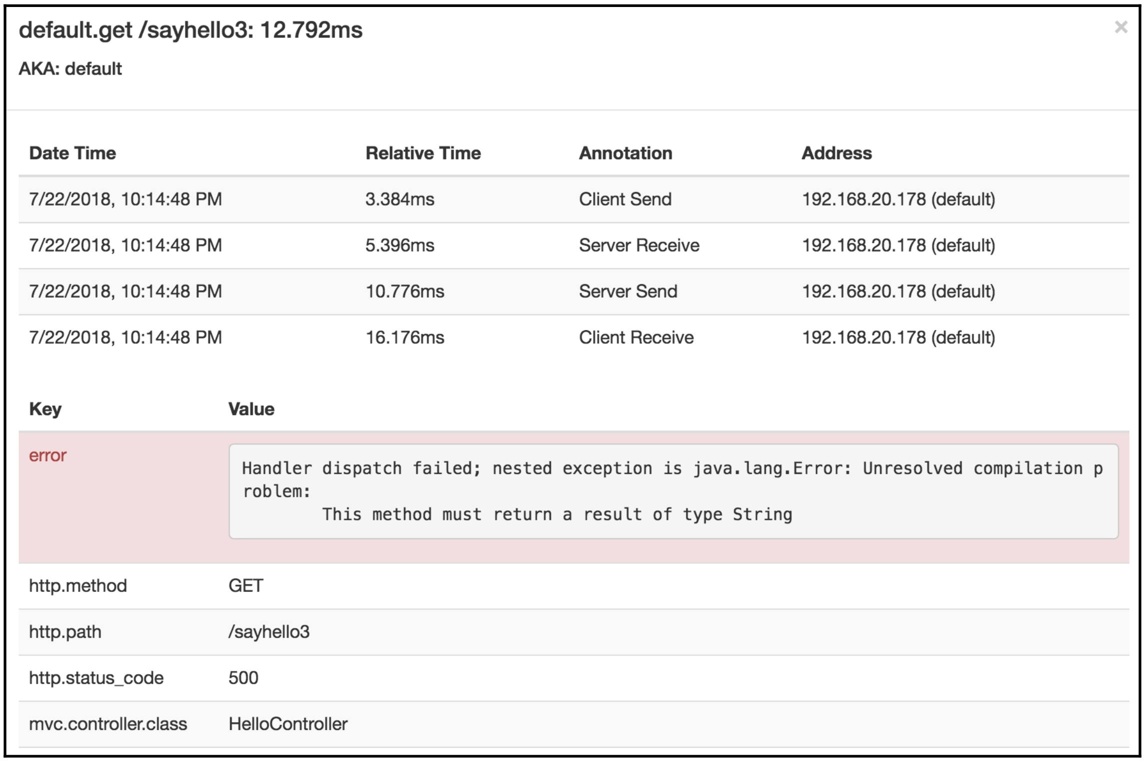
The following screenshot shows the Zipkin interface when service 3 is not responding properly:



Duration: 30.938ms Services: 1 Depth: 3 Total Spans: 3

```
Services         6.188ms     12.375ms     18.563ms     24.750ms     30.938ms
- default   30.938ms : get /sayhello
 - default     25.105ms : get /sayhello2
    default         12.792ms : get /sayhello3
```

It highlights the sayhello3 service to indicate there is a problem. In addition, you can click on each row and see the error details at each step. For example, the following lists the details of the error in service3:



default.get /sayhello3: 12.792ms

AKA: default

Date Time	Relative Time	Annotation	Address
7/22/2018, 10:14:48 PM	3.384ms	Client Send	192.168.20.178 (default)
7/22/2018, 10:14:48 PM	5.396ms	Server Receive	192.168.20.178 (default)
7/22/2018, 10:14:48 PM	10.776ms	Server Send	192.168.20.178 (default)
7/22/2018, 10:14:48 PM	16.176ms	Client Receive	192.168.20.178 (default)

Key	Value
error	`Handler dispatch failed; nested exception is java.lang.Error: Unresolved compilation problem: This method must return a result of type String`
http.method	GET
http.path	/sayhello3
http.status_code	500
mvc.controller.class	HelloController

This clearly shows that there was a developer error, and a new update has made the sayhello3 service uncompilable and hence not reachable. Zipkin helped us understand what and where the problem was with a few simple clicks.

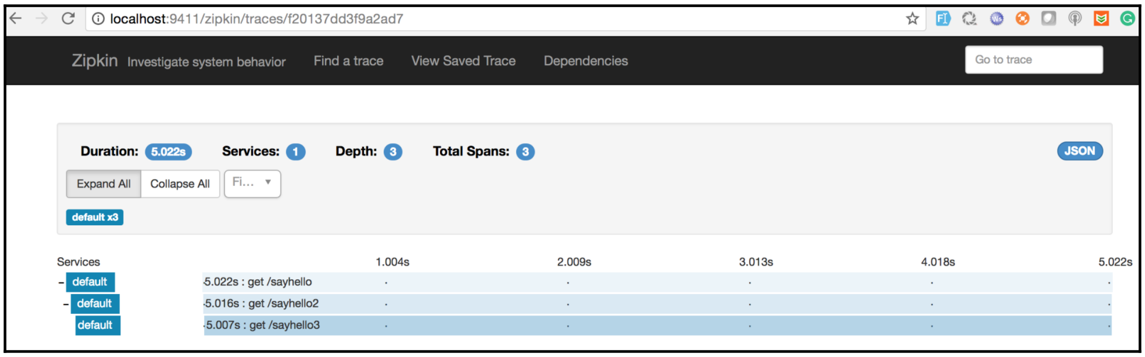
Case 2 – service responding slowly

Let's try another case. Here, the API is responding very slowly. To simulate that, let's say we introduce an artificial delay of five seconds in service3:

```
@GetMapping("/sayhello3")
public String sayHello()
{
 try
 {
  Thread.sleep(5000);
 }
catch (InterruptedException e)
 {
  e.printStackTrace();
 }
 return("Hello from service 3.");
}
```

When we call the sayhello service, we see it is responding slowly. But again, we are not sure what is making the service so slow to respond. Again, let's take a look at Zipkin to see whether it can help us understand where the problem is.

The following screenshot shows the Zipkin interface:

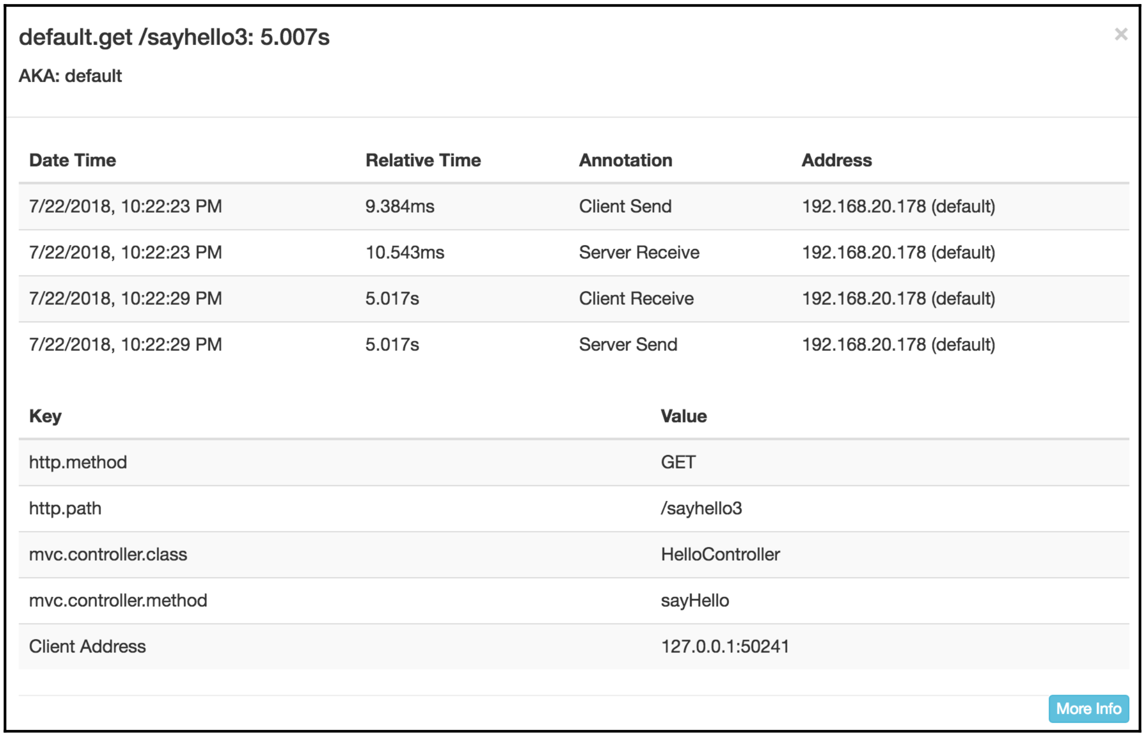


Duration: 5.022s Services: 1 Depth: 3 Total Spans: 3

```
Services              1.004s        2.009s        3.013s        4.018s        5.022s
- default     5.022s : get /sayhello
 - default    5.016s : get /sayhello2
   default    5.007s : get /sayhello3
```

This indicates the sayhello3 service has taken a good amount of time. To see more details, we can click on the third row for sayhello3.

The following screen appears when we click on the third row:



default.get /sayhello3: 5.007s

AKA: default

Date Time	Relative Time	Annotation	Address
7/22/2018, 10:22:23 PM	9.384ms	Client Send	192.168.20.178 (default)
7/22/2018, 10:22:23 PM	10.543ms	Server Receive	192.168.20.178 (default)
7/22/2018, 10:22:29 PM	5.017s	Client Receive	192.168.20.178 (default)
7/22/2018, 10:22:29 PM	5.017s	Server Send	192.168.20.178 (default)

Key	Value
http.method	GET
http.path	/sayhello3
mvc.controller.class	HelloController
mvc.controller.method	sayHello
Client Address	127.0.0.1:50241

We can see most of the time is being taken by service 3 and the response time of the other two services is in milliseconds. This helps us isolate the problem area, which is service 3 in this case.

We have seen how a tool such as Zipkin can help us track errors at the Microservice level. This can be very useful in a production environment when we face an issue and need to validate which service is in a problem state.

There can be cases where you do not want to apply to trace to all the request as of course tracing comes at a cost in terms of performance (though negligible, in a production environment you want to be extra careful). Zipkin comes with an option of tracing only a sample of requests, which is configurable, such as only tracing 10% of requests.

Tools for monitoring Microservices

We have already talked about a couple of tools, Zipkin and OpenTracing, that can help us to implement the monitoring of Microservices. We also looked at a detailed example of the use of Zipkin, which helps us trace error scenarios.

We will take a look at a couple of additional tools that can help us to monitor Microservices. We do not claim to cover all tools or recommend any tools to readers, as use of tools depends on the situation and personal preference. We are trying to get an idea at a high level about these tools and their uses.

Prometheus for monitoring and alerting

Prometheus is an open source monitoring and alerting tool. It gathers time series-based numerical data from the applications being monitored. Written in the GO language, the tool captures metrics in the metrics 2.0 format (`http://metrics20.org/`): the metrics have a name, a description, dimensions, and values. The only thing missing is a unit for the metrics.

Prometheus is used for Whitebox monitoring, where, the application being monitored is aware that it is being monitored. Endpoint-defining metrics are exposed over HTTP by the application. Prometheus uses various exporters that can share data with the server. One of the most widely used exporters is NodeExporter. When NodeExporter is run on a host, it will provide details on I/O, memory, disk, and CPU pressure. You can create exporters to monitor almost anything in the service, such as API calls, method calls, or database interactions.

The tool provides PromQL, a sophisticated query language to fetch time series data is stored.

The following diagram shows the high-level architecture of Prometheus:

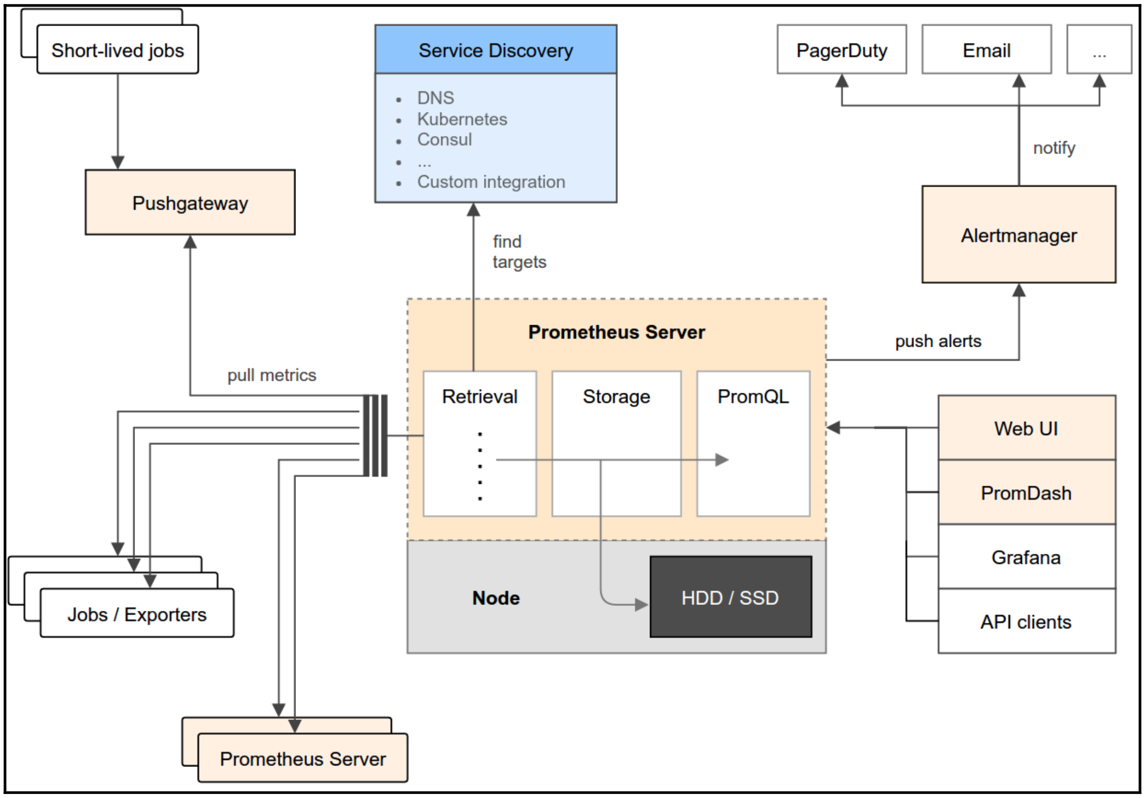


Image Source :
`https://prometheus.io/docs/introduction/overview/#architecture`

The architecture showcases the following components:

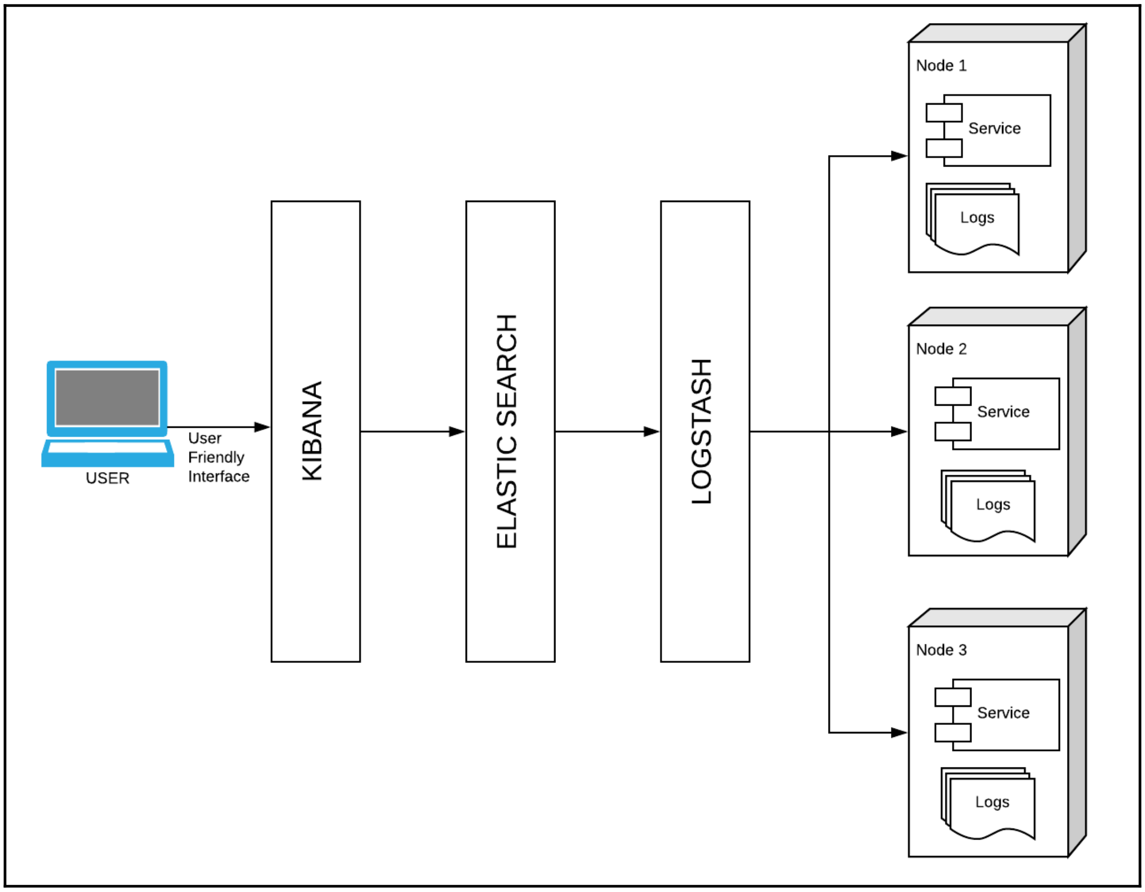
- **Prometheus server:** This server collects the metrics from applications and stores them locally. The Prometheus server works on the principle of scraping, that is, invoking the metrics endpoints of the various nodes that it is configured to monitor. It collects these metrics at regular intervals and stores them locally.
- **Push gateway:** There are the cases when an endpoint cannot be exposed by the application due to nature of its work, such as static jobs. The Push gateway captures the data, transforms that data into the Prometheus data format, and then pushes that data onto the Prometheus server.
- **Alert manager:** Can deliver alerts to multiple channels, such as SMS, email, and Slack, based on alerting rules.

Elasticsearch, Logstash, and Kibana (ELK)

ELK is a combination of three open source tools: **Elasticsearch**, **Logstash**, and **Kibana**:

- **Logstash** is an open source tool for collecting, parsing, and storing logs for future use.
- **Elasticsearch** is a search and analytics engine. It works on logs collected by Logstash.
- **Kibana** is a web interface that can be used to view data in a useful and appealing format.

The following diagram shows how the three tools work together:



As the preceding architecture explains, the three tools of ELK work together to fetch and showcase the analytics information to the end user. Logstash is responsible for fetching logs from a distributed system, where different Microservices might be deployed on different machines.

Once the logs are available, Elasticsearch helps to implement search and analysis capabilities. Finally, Kibana displays the data in various graphs or diagrams, which are more meaningful to the user and gives easily actionable items.

Considering more tools

We have talked about a few important tools for monitoring Microservices. There are additional tools one can consider based one's need. Splunk can be used for logging, indexing, and fetching information, somewhat similarly to ELK. Fluentd can be used to implement a logging layer. Logspout can be used to collect and forward logs. We already looked at Zipkin and OpenTracing for tracing; some similar tools are Appdash and Phosphor. We talked about alerting with Prometheus, but you can use other tools, such as PagerDuty, for alerting. For simple checks, such as service availability, try Pingdom.

In addition, if you are using a cloud service provider, such as Amazon Web Services, Google Cloud, or Microsoft Azure, you might get some tools out of the box, which will help you monitor and manage your Microservices.

A complete discussion on all these tools is outside the scope of this chapter, but the information shared should give you an idea of the core concepts and considerations you need to be aware of when you implement monitoring for Microservices-based applications.

Summary

In this chapter, we talked about monitoring and profiling Microservices. Monitoring Microservices brings its own challenges, as different Microservices might be deployed on separate machines, so we have to deal with a distributed infrastructure and a number of services. You need to keep tabs on how much time a service is taking to respond; you don't want your end user to wait too long for a response.

In addition, you need to understand which services are on a critical path, and which services are more important than others, so that we can take a call on the severity of the issue.

We talked about tools that can provide us the data required for analysis. We can use these tools to fetch data for historical analysis, and for health monitoring. The retention of data is another important factor. For how long do you want to keep the data? All these decisions one needs to take based on specific needs.

We have not yet talked about services deployed in containers, such as Docker and Kubernetes. Container-based deployments bring their own advantages and challenges. This is what we will be looking at in the next chapter.

9
Building, Packaging, and Running Microservices

The Microservices architecture emerged as a result of overall advances in software architecture and the build process. With Microservices, the responsibility for solving the problem domain is no longer delegated to a single monolithic system with many functionalities.

A Microservice is bound to a specific context in the problem domain. As a purely architectural approach, building a Microservice does not imply any way of assembling the application, building it, or running it. There are no technical implications whatsoever. In order to build a Microservice, the problem domain must be analyzed, then a set of mutually exclusive services is shaped and developed. It can be programmed in the same language as a monolith, built in the same way as a monolith and, of course, run in the very same manner. The Microservices internal architecture may be arbitrary, and the same architectural principles for building monoliths may be leveraged when building Microservices.

However, Microservices and the way applications are built nowadays allow strong paradigm shifts.The aim of this chapter is to provide the reader with a thorough understanding of the various possibilities of the Java Microservices shipment. The reader is able to recognize attributes of different ways of packaging Java applications. The goal of this chapter is to teach the reader to be able to evaluate the benefits and disadvantages of each and every Java application distribution model. Therefore, we will look at how packaging affects the process of the Microservice development, packaging, and distribution.

The following topics will be covered in this chapter:

- Introduction to Java Packaging
- Java EE MicroService Solution
- Deployment Architecture for Microservices

Introduction to Java Packaging

In the Java universe, there are three basic types of files:

- Class files
- Other applications' resources, such as scripts and images
- Meta-information files

Class files contain Java source code translated into bytecode. Bytecode is then executed by JVM. Such files are easily recognized by their `.class` suffix. The Java code is solely contained in those classes. However, a Java application may require additional resources during the application's runtime: websites may require images to be served to the client, or a simulation may require a file with an initial simulation state. Simply put, any non-Java-related file is considered to belong to the group of other application resources, with one exception: files that contain meta-information about the application itself. Security-related information, information about an application's version, or application configuration data are considered meta-information. Developers mostly influence class files and application resources, as these directly reflect the application being written. The application is a general term we are using, Microservices can also be considered as Java applications when it comes to packaging.

An application is a set of directories that contain classes and related resources. For an application's users, it is easier to obtain one archive with the whole deployable. There are more types of archives in the world of Java:

- **Java Archive** (**JAR**)
- **Java Web Archive** (**WAR**)
- **Java Enterprise Archive** (**EAR**)

A JAR is a basic form of packaging used in the Java universe. Other forms of packaging build on top of the JAR, extending functionality. A JAR contains all application classes and some meta-information, such as the application version. Also, the whole JAR, as well as any other archive, can be digitally signed using the **Public key infrastructure** (**PKI**). A JAR file can be considered as a ZIP file with a standardized inner layout. Compression is completely optional.

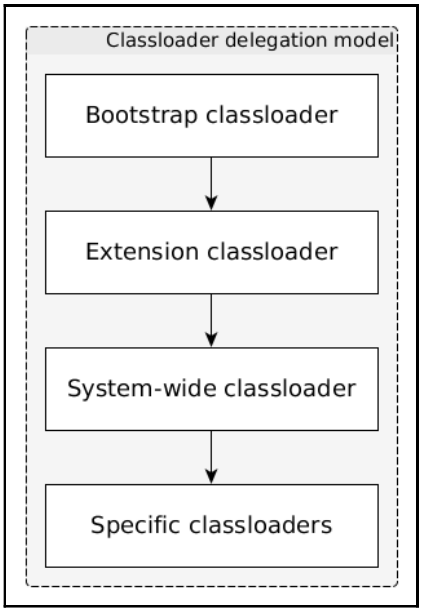
A WAR uses the same principle as a JAR. It was first introduced with the Servlets standard. The main benefit of introducing WAR files is new types of web-related content, such as servlets or web pages. A web archive may also contain multiple JARs. Commonly, these are libraries or different parts of the application.

The most complex and rarely used type of packaging is an EAR. An EAR file provides a means to bundle multiple WARs and JARs into a single package. An EAR provides means to define the deployment order of the archives contained in it.

Understanding Archives

The JAR and WAR files mostly do not come with a complete set of libraries required for code to run. In practice, programs rely on common libraries. In Java, the standard library is a perfect example. Java programs rely on the standard library to operate. Packages beginning with `java.*`, such as `java.io` or `java.lang`, are part of the standard library and are not bundled with the application. The separated distribution of common libraries is a general concept practiced for decades. Package managers in Linux operating systems only load shared libraries for each program once. When a program is installed, the package manager iterates over libraries required for the installed application to be present on the system and only download the missing ones. This way, the computer's persistent storage is not easily bloated with many instances of the same library. The same principle is applied in the Java world. If every Java application, including Microservices, depends on the standard library, there is no reason to distribute the standard library with the application, but instead install it once on the system Java applications are expected to run. This makes the Java-based application archives much smaller and faster to distribute.

The principle of the class loaders hierarchy in Java reflects the pattern of shared or common dependencies. Class loaders are special classes dedicated to loading the resources required for a Java application to run, including the application classes. There are more types of class loaders, each type being represented by one or more classes, as shown here:



Different classloaders load different parts of code from different places. There is no inheritance among classloaders, only delegation. The structure is represented by a directed acyclic graph. The principle of class loaders can be summed up in the following statements:

- Different resources are loaded by different class loaders.
- Different class loaders are invoked at different times.
- One type of class loader may be invoked zero or multiple times.

The bootstrap, also called the **primordial**, class loader is not distributed with the application. As the Java standard library is distributed as a separate package, the bootstrap class loader is also part of the Java Runtime Environment package. Other class loaders may be distributed with an application server or servlet container; some may even be part of the final application, if required.

The principle of shared dependencies can be generalized even further. Another place where sharing libraries is a big concern is memory management. For any program to be executed, it is first loaded from persistent storage into a significantly faster memory, nowadays usually a RAM. Different components of each application are loaded on demand, so-called **lazily**, that is loaded only when required. It is common for two or more applications to require the same library to be loaded in memory, and it is a common feature of the operating system's memory management to load the library only once. Unlike in the case of persistent storage, volatile memory is much more limited in space. As the operating system loads common libraries only once, the goal is for the Java runtime only to load the same libraries once.

Next, we will discuss fat packages and the creation of fat JAR.

Fat packages

Sometimes, the separation of resources is not required. In such cases, it is possible to reduce the number of layers used and simply join two or more layers into one. Historically, there are more use cases for this.

FatWAR packaging

FatWAR in the Java world are almost exclusively the domain of Spring Framework. In order to understand the reasons for the existence of FatWAR, we must understand the architectural decision made in the process of Spring Framework creation.

FatJAR packaging

In this chapter, the reasoning behind the FatJAR packaging style is explained. The road leading to FatJAR packaging is explained, as are the advantages and disadvantages of a FatJAR solution. In the world of Java EE Microservices, both pure Java EE and additionally Spring Framework is represented by a set of libraries. Even though Java EE gives us an option of separating the code and dependent libraries, in the end, both the API and the libraries with implementations contained inside are required for the application to run. In the previous era of Application servers, the APIs and their implementations were already contained within the application server. The applications deployed onto the application server did not carry such libraries inside. Those dependencies were provided by the application server as a runtime environment for the application. In fact, most modern application servers do not load all the libraries at startup, but leverage dynamic loading to create a rightsized environment that is a perfect fit for the application running on it, which also results in resource savings.

For historical reasons, an application server usually hosted more than one application. The process of extracting shared libraries into a common place led to various optimizations:

- Persistent storage space savings
- Memory savings
- Application-size shrinks
- Deployment time is reduced

A first observation is simple: the library only has to be distributed once. This paradigm is used across many fields in computer science and is generally considered a good practice. Linux package managers serve as a perfect example. Many user-space programs rely on the same libraries. Those are only downloaded once and are shared by all other dependent deployables.

Memory savings follow the same principle. When one or more deployment units require the same library, there is a possibility of only loading such a library into memory once. A common enhancement practiced by almost every application server is only to load the libraries truly required by applications deployed to a fast, nonpersistent memory. Dozens, or even hundreds, of applications, could be deployed onto a single application server. In general, if the number of applications deployed is more than one, cost savings occur by sharing a common library.

As a result, the size of the deployment unit containing the application, usually in the form of a Java Web Archive, is significantly lower. The application is only compiled against Java EE APIs. The Java EE APIs are only required during the process of compilation and are not included in the deployment unit. When no special third-party libraries were used, the deployment unit archive contained only classes with business logic and nothing more. The resulting size was in units of kilobytes. Not more. Naturally, when an application is deployed into an application server, the classloader, which happens to be specific to each application server, loads all the classes included in the deployment unit. Assuming loading a single class has a fixed computational complexity of $O(n)$, loading k classes results in a computational complexity of $O(k*n)$. In other words, the time required to load classes grows linearly with the number of classes included. Having the number of libraries loaded with each deployment reduced from potentially hundreds of megabytes to a few kilobytes significantly reduces the deployment time, usually down to dozens of milliseconds in the case of small changes.

With Microservices, each instance of a service exists in an isolated environment. Therefore, the situation with one application server having $n > 1$ applications deployed becomes obsolete. Each Microservice is packaged and scaled independently, with dedicated resources allocated for each instance, preventing resource sharing in ways an old-fashioned application server used to do.

Discarding the concept of separation of runtime dependencies and business logic represents a significant simplification. In the Java world, a self-sufficient JAR with all runtime dependencies packaged into one package is known as FatJAR. Often, alternative names, such as UberJAR, are used.

Java EE MicroService solutions

Java provides a lot of solutions that can help us develop, package, and run Microservices. We will now discuss a few solutions available for packaging and running Microservices.

OpenLiberty

OpenLiberty is a Java EE solution for running Microservices made by IBM. OpenLiberty was introduced as an evolution of the former WebSphere Liberty in 2017. As its name suggests, it is available under an open source license, without any warranties implied. IBM was quick to deliver basic support for crucial Java EE 8 parts, including Servlet 4.0, CDI 2.0, and JPA 2.2, even though the whole set of Java EE 8 specifications wasn't supported at the time. What really stands out is the support for the Eclipse MicroProfile specification in its latest version, 1.2, implying health checks, fail-safe mechanisms, circuit breakers, and configuration APIs are fully supported.

The following are key factors of OpenLiberty:

- **Homepage**: `https://openliberty.io/`
- Eclipse Public License 1.0
- MicroProfile 1.2 full support
- Java EE 7
- Partial support for Java EE 8

OpenLiberty is available in two flavors. First, it is available for download as a whole package, acting as an application server with all the dependencies in one place. Second, there is a plugin for both Maven and Gradle that downloads only the parts required by the actual application, producing a rightsized package.

One of the comparative advantages of OpenLiberty is fast redeployments. IBM designed OpenLiberty with fast redeployments in mind, resulting in almost instant code changes without the need to redeploy the application or even restart it.

The OpenLiberty Maven plugin

With the OpenLiberty Maven plugin, it is possible to manage compilation, deployment, and packaging by executing Maven goals. Besides other service-level tasks, the following maven goals are considered to be the pillars of Microservice handling with OpenLiberty:

- Generate a self-contained, executable archive with `liberty:package-server`.
- Start a rightsized `OpenLiberty.io` for automatic fast redeployments with `liberty:run-server`.
- Start or stop new OpenLiberty instance in a separate process with `liberty:start-server`.
- Deploy/undeploy Microservice with `liberty:deploy` and `liberty:undeploy`.

In order for `OpenLiberty.io` to provide all the functionality, these three steps must be taken:

1. Declare an OpenLiberty Maven parent.
2. Declare and configure the OpenLiberty plugin.
3. Create the OpenLiberty configuration.

All three steps are required. The first two steps are to be taken in Maven's `pom.xml` file. To fulfill the third step, creating a `server.xml` file with the OpenLiberty configuration is necessary. The OpenLiberty Maven parent is a convenient way to provide all the tooling required for the Maven plugin to work and not to force the developer to learn the internal mechanisms of OpenLiberty.

The code below shows a `<parent>` XML tag:

```
<parent>
    <groupId>net.wasdev.wlp.maven.parent</groupId>
    <artifactId>liberty-maven-app-parent</artifactId>
    <version>2.0</version>,
</parent>
```

At the time this book was written, the latest version of `liberty-maven-app-parent` was 2.0. However, it is anticipated that there will be a significant number of improvements introduced in the future. Using the latest version may be an advantage as it will provide the latest code with bug fixes.

As a second step, the OpenLiberty Maven plugin is declared and configured. In a typical Maven project, there is more than one plugin involved. In the following example, the OpenLiberty Maven plugin declaration is placed into the `<plugin>` XML tag, among other commonly used plugins, such as the compiler plugin or war-plugin:

```
<plugin>
    <groupId>net.wasdev.wlp.maven.plugins</groupId>
    <artifactId>liberty-maven-plugin</artifactId>
    <version>2.0</version>
    <configuration>
        <assemblyArtifact>
            <groupId>io.openliberty</groupId>
            <artifactId>openliberty-runtime</artifactId>
            <version>${openliberty.runtime.version}</version>
            <type>zip</type>
        </assemblyArtifact>
        <serverName>${project.artifactId}Server</serverName>
        <stripVersion>true</stripVersion>
        <configFile>src/main/liberty/config/server.xml</configFile>
        <packageFile>${package.file}</packageFile>
        <include>${packaging.type}</include>
        <bootstrapProperties>
            <default.http.port>${testServerHttpPort}</default.http.port>
            <default.https.port>${testServerHttpsPort}</default.https.port>
            <app.context.root>${project.artifactId}</app.context.root>
        </bootstrapProperties>
    </configuration>
    <executions>
        <execution>
            <id>package-server</id>
            <phase>package</phase>
            <goals>
                <goal>package-server</goal>
            </goals>
            <configuration>
                <outputDirectory>target/wlp-package</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The plugin introduces several configuration variables and behaviors. The behaviors are binded to various Maven goals. The preceding code provides a more advanced example of many useful configuration options OpenLiberty provides. There are reasonable defaults for most of the properties mentioned. The plugin declaration contains the following:

- The plugin itself, including a group identifier, artifact identifier, and version
- Binding of the package-server goal to the standard Maven package goal
- The path to the server configuration file
- Additional properties, including server name or startup port to bind to

Plugin declaration is a self-explaining step and easy for readers with previous knowledge of Maven. In the `<execution>` section, a Maven goal named `liberty:package-server`, introduced by the OpenLiberty Maven plugin is bound to the standard Maven's package goal. As the `liberty:package-server` goal produces a self-contained application with a rightsized OpenLiberty runtime inside, binding it to Maven's package goal ensures that an up-to-date version of such self-contained Microservices is produced every time a standard WAR is produced. This enables developers to automate the OpenLiberty solution creation.

If we remove this binding, a manual invocation of `liberty:package-server` is required to produce a self-contained, distributable application based on OpenLiberty. It is left to the user's discretion whether they want to go for automated or manual invocation of package-server.

Example command:

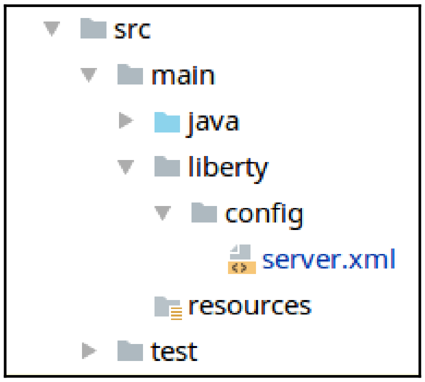
```
mvn liberty:package-server -DserverHome=/path/to/home -
DserverName=somenamehere
                              -DpackageFile=/filename.zip
```

Configuring OpenLiberty

OpenLiberty provides a rightsized server. In order for the Maven/Gradle plugin to incorporate the libraries required for the application, a configuration file named `server.xml` is introduced. A standard Java project layout, as defined by Maven, contains two folders in the `main/` folder:

- Java folder with `.java` files
- Resources folder with files related to the application, but not containing code

When OpenLiberty is used, a third directory named `liberty/` must be placed into the `main/` folder. Within the `liberty/` directory, there is one more subdirectory, named `config/`, to be found, which is where the OpenLiberty `server.xml` configuration file resides. The hierarchy can be observed in the following filesystem tree:



A minimalistic version of `server.xml` contains only a list of features required to be present for the application during its deployment. For a simple RESTful Microservice, there is only one feature to add. If more features are required, for example CDI for dependency injection or JPA for object-relational mapping and persistence, then including more `<feature>` tags with feature names inside is the obvious choice. Each feature contains the name of the feature complemented with a version. An exhaustive list of existing features can be found on OpenLiberty's GitHub page: `github.com/OpenLiberty/open-liberty`.

```
<server>
  <featureManager>
      <feature>jaxrs-2.0</feature>
  </featureManager>
</server>
```

The destination of the `server.xml` file can be changed in Maven's `pom.xml` where the `liberty-maven-plugin` is configured. Inside the `liberty-maven-plugin` configuration, there is a `<configFile>` tag to be found. The value used for this book corresponds with the path described in this chapter and is configured as `<configFile>src/main/liberty/config/server.xml</configFile>`.

The Weather Microservice with OpenLiberty

In this chapter, a simple temperature Microservice already known from previous chapters is going to be implemented, demonstrating the simplicity of creating a Microservice with IBM's OpenLiberty. The Microservice built is available in this book's source code for a review and trial.

To create the Weather MicroService with OpenLiberty, you only need Java EE's JAX-RS and OpenLiberty configuration, which is described in this chapter. The code for the temperature Microservice created in the `Chapter 2`, *Creating your first Microservice* may be reused without any modifications whatsoever. First, you need to create the configuration:

```
package com.packtpub.Microservices.openliberty;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/weather")
public class WeatherMicroservice extends Application {

}
```

This configures the basic path for a weather service. The next step is to implement a RESTful endpoint, introduced in the second chapter, returning artificial average temperature whenever HTTP GET is issued upon the endpoint. Only a Java class decorated with annotations from the `javax.ws.rs` package is required:

```
package com.packtpub.Microservices.openliberty;

import com.packtpub.Microservices.domain.weather.Temperature;
import com.packtpub.Microservices.domain.weather.TemperatureScale;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

/**
 * RESTful resource providing information about city's temperature
 */
@Path("/temperature")
public class TemperatureResource {

    /**
     * Provides average temperature from all the city's sensors. The
temperature
     * is artificial.
```

```
         *
         * @return {@link Response} with constant temperature
         */
        @GET
        @Produces(MediaType.APPLICATION_JSON)
        public Response getAverageTemperature() {
            Temperature temperature = new Temperature();
            temperature.setTemperature(35D);
            temperature.setTemperatureScale(TemperatureScale.CELSIUS);

            return Response.ok(temperature).build();
        }
    }
```

Both classes (the configuration and the endpoint itself) are located in the `com.packtpub.Microservices.openliberty` package. The package name is completely arbitrary and does not influence the functionality of the resulting OpenLiberty Microservice.

In order to run the Microservice in a rightsized OpenLiberty runtime environment, two Maven commands must be issued:

```
mvn package

mvn liberty:run-server
```

Or simply execute the command `mvn` package `liberty:run-server`. The Maven package builds and packages the Java EE Microservice with the temperature endpoint inside and produces a WAR. Such a WAR can be deployed to any existing application server and is not modified in any way. However, as the OpenLiberty Maven plugin is attached to the package phase of the build process, a self-contained zip file with both Microservice and OpenLiberty inside is generated.

During development, running a self-contained application over and over again is repetitive and may affect the developer's performance. The role of `liberty:run-server` is to create an instance of OpenLiberty and keep it running during the whole development process. During startup, OpenLiberty scans for archives created with the `mvn` package. Even when the Microservice is not yet packaged, once OpenLiberty is running, it scans the `target/` folder in Maven's hierarchy for new applications to deploy. Once the application is deployed, it scans for changes in classes and resources. Once a class is recompiled, OpenLiberty detects the changes and instantly applies the new classes only. No complete redeployment happens. This way, code changes are visible almost instantly. It usually takes only dozens of milliseconds for OpenLiberty to apply the changes. From a developer's perspective, the process is instant.

After OpenLiberty is started and the `temperature` Microservice is packaged, simply invoking the `http://localhost:9080/10-OpenLiberty.io/weather/temperature` endpoint results in an immediate response. By default, OpenLiberty binds to localhost on port `9080`. The port can be explicitly configured in the OpenLiberty Maven plugin configuration, as demonstrated in this chapter. The application context is by default the name of the `application/` Microservice artifact being deployed, without any suffix. It can be controlled both in the `server.xml` configuration or by means of the Maven plugin configuration. The expected output is demonstrated in the following code block:

```
{
"temperature":35.0,
"temperatureScale":"CELSIUS"
}
```

OpenLiberty represents a way of rightsizing the application runtime. It may no longer be considered a pure application server only. If required, OpenLiberty can act as an application server, hosting a number of applications at once, providing many features up to the point where it is a full-blown Java EE 8 Application Server.

The Gradle plugin

The primary build tool used in this book is Maven. With Gradle getting more and more popular, it should be mentioned that OpenLiberty also provides full support for Gradle. The OpenLiberty Gradle plugin can be used in the following ways:

- Apply the OpenLiberty Gradle plugin to the `project`.
- Define the `OpenLiberty.io` runtime as a runtime dependency.
- Instruct Gradle to invoke OpenLiberty with common Gradle goals.

It all starts the standard Gradle way, by applying a plugin:

```
apply plugin: 'liberty
```

The plugin must be made available to the build strict. A typical Gradle build script block contains a Maven central repository, as demonstrated here. In addition, the OpenLiberty's Gradle plugin is applied as a dependency on the classpath:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'net.wasdev.wlp.gradle.plugins:liberty-gradle-plugin:2.1'
    }
}
```

OpenLiberty does not force the resulting artifact to contain OpenLiberty-specific libraries. The OpenLiberty plugin introduces a special dependency scope, named `libertyRuntime`. The name is self-describing; it tells the reader such dependencies are only bundled inside the OpenLiberty runtime. Such a dependency is often accompanied by other dependencies. In the case of Java EE 8, the whole API may be included as a provided dependency:

```
    dependencies{
        providedCompile 'javax:javaee-api:8.0'
        libertyRuntime group: 'io.openliberty', name: 'openliberty-runtime',
    version: '[17.0.0.4,)'
    }
```

As in the case of Maven, OpenLiberty offers a convenient way to configure default properties. The properties are an exact copy of those described in the Maven section. The server name, default ports, context root, or name of resulting archive, everything may be reconfigured. Java EE 8 projects tend to evolve rapidly, so are OpenLiberty plugins for Gradle and Maven. For a complete list of features, configurables, and settings, please visit the reference guide at `https://openliberty.io/docs/`.

```
    liberty {
        server {
            name = "$OpenLiberty Server"
            configFile = file("src/main/liberty/config/server.xml")
            bootstrapProperties = ['default.http.port': 8080,
                                   'default.https.port': 8443,
                                   'app.context.root': ${appName}]
            packageLiberty {
                archive = "$buildDir/${appName}.zip"
                include = "usr"
            }
        }
    }
```

To run the application, the following Gradle tasks similar to Maven's goals are available:

`libertyStart`

`libertyStop`

WildFly Swarm

With its recent update, WildFly Swarm has been renamed Thorntail. A among developers, the name of WildFly Swarm is still commonly used. WildFly Swarm represents an excellent approach to create deployables for Java EE Microservices. WildFly Swarm stands on the shoulders of a WildFly Application server, which itself stands on the shoulders of one of the most famous names in Java EE world: JBoss. As explained in first section of this chapter, runtimes for Microservices in the Java EE world, be it any product (including Spring Boot), are only repacked application servers with just enough of the runtime dependencies the actual application requires. Additional services, plugin support, cloud integration, and available support make all the difference. And WildFly Swarm offers an excellent package.

The release cycle of WildFly Swarm is considerably faster than a full-blown WildFly Application service. There are monthly updates to be found, not only containing bug fixes, but coming with many improvements and support for new features. In general, WildFly is evolving very rapidly. In the MicroService environment, the WildFly swarm makes it easy to use an application throughout different environments, as it automatically configures thread pool sizes at startup based on available resources. At startup, a message revealing the actual configuration can be found:

Worker 'default' has auto-configured to 16 core threads with 128 task threads based on your 8 available processors.

In this case, WildFly detected eight physical cores with 18 threads available, resulting in 128 threads available in the default pool. This way, neither developers nor administrators have to worry about resource allocation configuration, as the application is deployed in vastly different environments.

The WildFly Swarm generator

It is a usual feature of Microservice-related products in the Java EE world to provide a project generator. WildFly Swarm also provides one. One thing to notice is the number of services provided, which is outstanding. A unique feature is automatic dependency detection. WildFly plugins automatically scan the project's source code and include dependencies required at runtime, without any need for explicit specification.

Here is a link for the generator tool:

`https://thorntail.io/generator/`

Besides traditional Java EE dependencies, there are more or less expected dependencies. Support for RedHat's Cloud is expected, as such products are a way for companies to bring new users to their Cloud ecosystems, as Pivotal does with Spring, and IBM with OpenLiberty, by providing an easy means of integration. RedHat's name for the cloud is OpenShift. In the case of WildFly, OpenShift support is excellent, including in-cloud service discovery. Another very strong aspect of WildFly Swarm is the Apache Camel integration, with many modules for practically every massively used technology, including databases. Database support includes standard JPA datasources, extended with OrientDB, MongoDB, Cassandra or several other in-memory databases. WildFly Swarm also provides support for MicroProfile.

The WildFly Swarm dependencies are categorized by their level of maturity. Stable dependencies are tested and ready to use in a production environment. Most of the dependencies are this mature. The other two levels for dependencies are unstable and experimental. These are common found for new and leading-edge functionality.

The WildFly Swarm Maven plugin

WildFly Swarm offers a Maven plugin solely. At the time of writing, there is no Gradle plugin available for WildFly Swarm. In general, Maven projects may be generated by means of a project generator. In this section, a very minimalistic temperature Microservices with minimal runtime using WildFly Swarm is demonstrated. Also, Swarm's ability to detect dependencies is leveraged to produce a project with minimal configuration overhead.

The first step is to create a brand new Java EE project with Maven, using the `war` packaging:

```
<packaging>war</packaging>
```

In Maven's properties section, a version of WildFly is defined as a project-wide variable. Such an approach is advantageous as version upgrades would mean minimum change in Maven :

```
<properties>
    <version.thorntail>2.2.1.Final</version.thorntail>
    <!-- Other properties omitted -->
</properties>
```

Wildfly Swarm Maven dependencies are contained in a single **Bill of Materials** (**BOM**). All the dependencies are defined in a single entry, using the previously-defined version property:

```
<dependencyManagement>
    <dependencies>
        <dependency>
           <groupId>io.thorntail</groupId>
           <artifactId>bom-all</artifactId>
           <version>${version.thorntail}</version>
           <scope>import</scope>
           <type>pom</type>
           </dependency>
    </dependencies>
</dependencyManagement>
```

The WildFly Swarm Maven plugin is required to be activated explicitly. In the following example, a standalone executable artifact is created whenever the package goal is invoked. This also includes the install goal. The standalone artifact is to be found in `{project-root}/target/ folder`. It always has a suffix of `swarm.jar`. Such a Microservice only requires a standard JDK to run, everything else is included inside the artifact. A simple `java -jar application-name-swarm.jar` is enough for the Microservice to run:

```
<build>
    <finalName>wildfly-swarm-temperature-Microservice</finalName>
    <plugins>
        <plugin>
            <groupId>io.thorntail</groupId>
            <artifactId>thorntail-maven-plugin</artifactId>
            <version>${version.thorntail}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>package</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The structure of project dependencies is simple. The Java EE 8 API as a provided dependency is standard and expected. The temperature Microservice used throughout this book requires **Plain Old Java Objects** (**POJO**) contained in the common module and will probably not be part of the reader's Microservice. These are simply extracted to a separate module for readability and easy reuse. Note that there no WildFly Swarm dependencies are explicitly declared. WildFly Swarm is able to automatically detect the required dependencies:

```
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>8.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>common</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>
```

The resulting Maven's `pom.xml` file used in the example project is demonstrated as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>javaee8-Microservices-book</artifactId>
        <groupId>com.packtpub.Microservices</groupId>
        <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>10-wildfly-swarm-temperature</artifactId>
    <packaging>war</packaging>

    <properties>
        <version.thorntail>2.2.1.Final</version.thorntail>
        <!-- Other properties omitted -->
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <failOnMissingWebXml>false</failOnMissingWebXml>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>io.thorntail</groupId>
                <artifactId>bom-all</artifactId>
                <version>${version.thorntail}</version>
                <scope>import</scope>
                <type>pom</type>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <finalName>wildfly-swarm-temperature-Microservice</finalName>
        <plugins>
            <plugin>
                <groupId>io.thorntail</groupId>
                <artifactId>thorntail-maven-plugin</artifactId>
                <version>${version.thorntail}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>package</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>8.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>common</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>

</project>
```

Before the application is run, a RESTful endpoint with JAX-RS inside is added. First, a simple JAX-RS configuration is required. This empty configuration class that extends the `javax.ws.rs.core.Application` class uses the `@ApplicationPath` annotation from the same package to configure a basic Microservice context root:

```
@ApplicationPath("/weather")
public class WeatherMicroservice extends Application {

}
```

Finally, the static `temperature` service is added:

```
package swarm;

import com.packtpub.Microservices.domain.weather.Temperature;
import com.packtpub.Microservices.domain.weather.TemperatureScale;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/temperature")
public class TemperatureResource {

    /**
     * Provides average temperature from all the city's sensors. The
temperature
     * is artificial.
     *
     * @return {@link Response} with constant temperature
     */
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAverageTemperature() {
        Temperature temperature = new Temperature();
        temperature.setTemperature(35D);
        temperature.setTemperatureScale(TemperatureScale.CELSIUS);

        return Response.ok(temperature).build();
    }
}
```

Automatic dependency-discovery may be the observer when invoking the `wildfly-swarm:run` Maven goal. In the case of the sample MicroService, there is only one REST endpoint using JAX-RS to expose itself. Therefore, WildFly correctly detects JAX-RS. Undertow is used to serve HTTP requests. It is worth mentioning that Undertow is a high-performance product, dominating many benchmarks. The rest are services required to run. For example, logging functionality is used even at the application start itself:

```
INFO: Installed fraction: Logging - STABLE
org.wildfly.swarm:logging:2018.3.3
INFO: Installed fraction: Elytron - STABLE
org.wildfly.swarm:elytron:2018.3.3
INFO: Installed fraction: JAX-RS - STABLE
org.wildfly.swarm:jaxrs:2018.3.3
INFO: Installed fraction: Undertow - STABLE
org.wildfly.swarm:undertow:2018.3.3
```

The default context root is set to `/` and the default port that WildFly Swarm listens on is `8080`. Therefore, the example temperature endpoint can be found under the `http://localhost:8080/weather/temperature` URL.

As additional functionality is plugged-in, WildFly will automatically detect it and bundle corresponding modules. This approach works flawlessly with the Java EE functionality. A sample Microservice that provides a static temperature measurement is available in the code samples bundled with this book.

HollowJAR

Not only in the of Microservices, the actual business logic changes often, but the libraries not related to the business logic do not. Such a concept is very well-known in the domain of operating systems, where common libraries are not only installed once to conserve space on the persistent storage (typically a hard drive) but are also loaded once into memory to achieve the same amount for savings. Java EE implementation libraries, MicroProfile libraries, or even Spring Framework libraries are in the very same relation to Java Microservices or applications in general. These common libraries, changes occur much less often than changes that are done in the internal business logic of an application built by using the common libraries. The libraries are updatable, interchangeable, and distributable in a complete separation from the actual Microservice.

In RedHat, they made it possible for a Java EE Microservice to create a separate runtime environment and only deploy the application to that environment. This way, one of the biggest disadvantages of monolithic solutions is removed:

- Fast redeploys

- Small distribution overhead
- Layering principle is not broken
- Fast changes in Docker images

The libraries representing the actual runtime environment for a Microservice are hollow, as there is no Microservice or application added to the JAR. This is where the name hollow jar originates. WildFly Swarm makes it easy to create such artifacts with just enough dependencies to run a Microservice, but without the actual Microservice inside. To produce a HollowJAR with WildFly Swarm, simply set the `-Dswarm.hollow=true` Maven property before the application is built by Maven. Or add the `<hollow>true</hollow>` option to the WildFly Swarm plugin configuration in `pom.xml`. WildFly Swarm will detect this property and produce a separate JAR with the `hollowswarm.jar` suffix in Maven's `target/` folder.

In order to deploy an application to the hollow environment, simply pass the application's name as an argument:

```
java -jar weather-service-hollowjar.jar weather-service.war
```

Payara Micro

Traditionally, Payara is an application server derived from Glassfish, the reference implementation of Java EE standards. The latest version to officially support Java EE 8 is Payara 5. As Payara is directly derived from Glassfish, it is quick to implement new functionalities defined in various standards.

Payara's opinionated approach to Java EE Microservices is named Payara Micro. The very basics of Payara's approach are the same as Spring Boot, WildFly Swarm, or OpenLiberty: to repackage an application server and omit functionality not required in the modern environment of Microservices, making the resulting package thinner. However, Payara Micro lacks the feature of rightsizing. Instead, all the functionality is packed in a single artifact, resulting in size under 70 MB. Payara Micro supports Java EE 8, together with the latest MicroProfile specification.

There are three major ways of running Payara Micro:

- Start the Payara Micro instance, then perform the deployment from a command line.
- Use programmatic startup, and run and configure Payara Micro at application startup.
- Create an Uber Jar with Payara Micro and the Microservice/application.

The Payara Micro Maven plugin

The Maven plugin is capable of automatically downloading, starting, and stopping a managed instance of Payara Micro. Besides that, it is also able to automatically deploy Microservices built with Maven. The creation of Payara Micro Uberjar is also one of the capabilities of the Maven plugin.

As a good practice, before the plugin declaration itself, create a build-wide property that contains the plugin's version.

The following code shows properties tab for Maven:

```
<properties>
<payaramicro.maven.plugin.version>1.0.0</payaramicro.maven.plugin.version>
    <!-- Other properties omitted -->
</properties>
```

As a next step, the declaration of the plugin itself in `pom.xml` is necessary. This is also the final step. Standard Maven requirements for a plugin, such as a group identifier and artifact identifier, distinguished by a plugin version, are necessary. Everything happens automatically. The only configuration required is to tell Payara to deploy WAR files on startup. The execution phase is set to package in order to automatically generate an UberJar with both the Microservice and Payara inside every time the Maven package or install goals are executed:

```
<build>
    <plugins>
        <plugin>
            <groupId>fish.payara.maven.plugins</groupId>
            <artifactId>payara-micro-maven-plugin</artifactId>
            <version>${payaramicro.maven.plugin.version}</version>
            <configuration>
                <deployWar>true</deployWar>
            </configuration>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>bundle</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

All WAR are bundled and deployed automatically with such settings. If a WAR is required, instructing Maven to use the WAR packaging is necessary:

```
<packaging>war</packaging>
```

Dependencies are standard. The Java EE 8 API as a provided dependency is required for the sample Weather Microservice used throughout this book, as well as the common POJOs used for this project specifically. The final POM with the preceding dependencies is demonstrated as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>javaee8-Microservices-book</artifactId>
        <groupId>com.packtpub.Microservices</groupId>
        <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>10-payara-micro</artifactId>
    <packaging>war</packaging>

    <properties>
<payaramicro.maven.plugin.version>1.0.0</payaramicro.maven.plugin.version>
        <!-- Other properties omitted -->
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <failOnMissingWebXml>false</failOnMissingWebXml>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>fish.payara.maven.plugins</groupId>
                <artifactId>payara-micro-maven-plugin</artifactId>
                <version>${payaramicro.maven.plugin.version}</version>
                <configuration>
                    <deployWar>true</deployWar>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>bundle</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>8.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>common</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>


</project>
```

In the `target/` folder created by Maven, there is the WAR file with the Microservice. Additionally, with the plugin settings demonstrated, there is an UberJar with the `-microbundle.jar` suffix. In the case of the sample project, its name is `10-payara-micro-1.0-microbundle.jar`. The sample project with Payara Micro is, of course, available in the package bundled to this book.

The UberJar may be executed manually by using the CLI. To do this, simply run the UberJar as any other Java application: `java -jar application-name-microbundle.jar`. The sample application is run by issuing the `java -jar 10-payara-micro-1.0-microbundle.jar` command. Alternatively, the `payara-micro:start` Maven goal can be used.

The started Microservice binds itself to port `8080`, using the name of WARS as application context roots. During startup, Payara Micro informs what all internal Microservices are deployed. The simple Weather Microservice that provides an arbitrary `temperature` is available at `/weather/temperature`, as in other projects in this book. Therefore, the resulting URL for the Weather Microservice is `http://localhost:8080/10-payara-micro-1.0/weather/temperature`. Sending an HTTP GET request to this URL returns an arbitrary temperature:

```
Deployed: 10-payara-micro-1.0.war ( 10-payara-micro-1.0.war war /10-payara-
micro-1.0 )
Payara Micro URLs
http://localhost:8080/10-payara-micro-1.0
]]
```

Payara Micro will download the latest stable release by default. Alternatively, the plugin can be pointed to a local instance of Payara Micro, or instructed to download a different version, if required. To instruct Payara Micro to download and manage a specific version, use Maven's standard plugin configuration block:

```
        <artifactItem>
            <groupId>fish.payara.extras</groupId>
            <artifactId>payara-micro</artifactId>
            <version>4.1.1.171</version>
        </artifactItem>
```

In the rare case that there is a Payara Micro instance available on the filesystem and intended to be used, an absolute path to the Payara MicroJAR can be specified as well:

```
<payaraMicroAbsolutePath>/path/to/payara-
micro.jar</payaraMicroAbsolutePath>
```

The Payara Micro UberJar

Inside the Payara Micro UberJar, there is a folder named `MICRO-INF/`. The configuration of Payara Micro, as well as Microservices deployed at startup, are present in this folder. And much more. A typical structure of this folder is demonstrated as follows:

```
.
├── classes
├── deploy
├── domain
├── lib
├── payara-boot.properties
├── post-boot-commands.txt
├── pre-boot-commands.txt
└── runtime
```

The names are self-descriptive. Microservices to be deployed are placed in the `deploy/` folder. There may be additional libraries used by those services. Those are placed in the `lib/` folder and typically configured in the Payara Micro Maven plugin configuration. The `runtime/` folder is of no interest to developers, as it contains implementations of the core functionalities Payara uses. Commands to invoke prior to boot and after the server has booted are placed in separate text files.

The Deployment Architecture for Microservices

A partial shift to the Microservices architecture in the world of enterprise Java, either driven by serious needs or by a desire to try something new, changed the way applications, now specialized as Microservices, are handled. In the old world, a curated set of common runtime libraries required for a usually monolithic application was represented by an application server. From a developer's point of view, an application server may have been viewed as a simple collection of Java libraries that implements a standardized API a developer could always rely on being there. Application servers provide much more functionality than that, such as the following:

- Clustering
- Load-balancing
- Fault-tolerance
- Diagnostics
- Security

Java applications could be automatically distributed among cloud nodes, while the application servers resolved load-balancing issues, reported errors, and operations were able to administrate all of the functionality easily.

In a Cloud environment, these functionalities, among many others, are usually moved to another layer. Even end-to-end security is often not left for the application server to manage. Load-balancing and clustering are managed by tools specific to the chosen cloud environment. In a stateless world, where the number of active nodes changes rapidly based on current load, such functionality is obsolete. There are built-in tools to perform health checks, handle TLS/SSL and provide fault-tolerance monitoring or even testing. Docker especially has taken the role of a universal container handled by many cloud environments with ease. Applications are just deployed to the cloud and scaled in the cloud. And the cloud environment is completely abstracted from the application by thin, self-managing wrappers, which are mostly Docker containers. However, moving solutions from these problem domains to outside of the application server does not mean the problems are gone, only that they're handled by different code on a different layer.

As applications, and especially Microservices, are deployed and scaled independently (especially in a cloud environment), the need for a big, heavy application server that is ready to host multiple applications at once, while being able to connect and maintain a connection with a cluster of other servers, support replication, and security realms is gone. Likewise, replicating it multiple times across the cloud is gone. Having just the amount of functionality that is required for an application to run is an important optimization.

However, looking at application servers and getting an impression of them being big, slow to start, and difficult to manage may be just a wrong impression. First, application servers are smart and use lazy loading. They load only the libraries required to run the deployed applications into memory; not everything is loaded at startup. This concept of lazy loading may result in the same or only slightly higher memory usage than a rightsized runtime environment. There may be situations where an application server, even though considered big and heavy, may eventually save resources. Having a single physical machine with many applications or Microservices deployed, as those do not require much computational power, requires only one application server to be up and running. When compared to the memory footprint and the footprint on a persistent storage of multiple Microservices, each running in their own dedicated environment, an application server might actually win in absolute numbers.

Solutions such as Spring Boot or OpenLiberty may reach down to 15 megabytes of artifact size and require approximately 25-30 MB of memory on startup. On the other hand, a full-blown application server with all the functionality is often slightly more than 100 MB on the hard drive and takes about 40-50 MB of memory on startup with the same application deployed, due to lazy loading. However, such low numbers apply to rightsized solutions only in the case of the simplest applications. In the real world, Microservices use databases, thread pool management, and bean pooling, as well as publishing REST APIs and generating logs, and many other functionalities. The more functionality it has, the less difference can be measured in size between a rightsized solution and an application server, as we end up creating a MicroService which more or less provides functinality of a complete application.

An application server also held on the principle of layering. The need for layers is not gone in the world of Microservices. In fact, it is more essential than ever. A simple yet very important piece of advice for developers is to seek vendors who do not break the principle of layering. Forcing developers to restart the whole environment due to changes in one class of one deployment unit may be considered poor engineering compared to standards used in the rest of the industry for several years. Being forced to upload dozens or hundreds of megabytes into Docker hub due to a small change in a small Microservice is not ideal either.

Pure Java EE solutions usually respect layering principles or even provide multiple ways for fast redeployments and tiny changes. OpenLiberty has a very clever class loader and Maven/Gradle plugins, making it easy for developers to see changes instantly during the development phase. WildFly Swarm offers hollow JARs, which are rightsized exactly for the Microservice. Payara Micro also acts as a separate runtime for applications/Microservice, event if not rightsized. The situation is a little less optimal with Spring Boot; running and stopping a Spring Boot Java Archive may be time-consuming. However, Spring offers a solution named Spring Loaded (`github.com/spring-projects/spring-loaded`). This is an open source project that enables developers to reload classes at runtime. It is not as advanced as other commercial alternatives, yet it rapidly reduces the number of restarts required, reducing non-productive time.

The Java EE solutions for the new era of Microservices are very mature, including Spring Boot. There is no longer any need to explicitly declare every runtime-required library. There are mature project generators and plugins to assist in creating self-contained Microservices during the build phase. Usually, the service providing vendors include packages to make deployment in their very own cloud environment, which is a factor to consider in the process of choosing the right solution. This way, even when programming against a fully-standardized Java EE environment, some amount of vendor lock may be introduced.

Summary

In this chapter, we looked at archive-generation in Java. Then, we discussed in detail of some of the solutions available to run and manage MicroServices in Java. We had a detailed discussion about OpenLiberty, Wildfly Swarm, and Payara Micro. Finally, we discussed the changes being observed in deployment architectures due to the popularity of MicroServices.

In the next chapter, we will cover the documenting and testing aspects of MicroServices.

10
Documenting and Testing MicroServices

In this chapter, we will talk about two important aspects of a Microservices-based application: documentation and testing. The documentation tells us what this service would do and how it can be used. Without this information, it is not possible for an end user to take advantage of the services you created.

The second aspect we will cover in this chapter is testing. Testing, as we know, is an important factor for any application code you write. You should never ship untested code. The age-old golden rule of software engineering, or any engineering for that matter, is to catch the errors early in the development, as the cost to fix things increases over time due to the impact a change will have on the overall application.

We will cover the following topics in this chapter:

- Documenting Microservices
- Swagger
- ApiDoc
- Additional Documentation Frameworks
- Testing Microservices
- Unit Testing
- Integration Testing
- Service Testing
- End-to-end Testing
- Exploring what to Test

Documenting Microservices

Documentation is often a neglected topic in the software world. In a hurry to deliver the end product to the user or get the applications deployed, we tend to ignore the importance of documentation, as it is not something that the end user will directly view. But as the application grows, we start feeling the pain of missing documentation. Imagine you are given a code without any documentation: it will take you more time to go through the code and understand it. Whereas if code, classes, and methods are properly documented, it becomes much easier to understand the code. If you already have experience with Java, you've probably used JavaDocs, which is a very strong and easy way to document code.

We will not get into the details of JavaDocs here, instead we'll focus more on the Microservices documentation, and particularly on API-level documentation, which is important, as often the users of these APIs are not the same developers who have implemented the APIs. These will be used by frontend developers or other teams using your API, so we need ways in which we can convey how to use a Microservice API properly to the teams and fellow developers.

Let's start with a simple calculator Microservice. Say we are exposing four basic operations: add, subtract, multiply, and divide, as follows:

```
package com.packt.Microservices.calculator;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class CalculatorController
{
@GetMapping("/add")
 public int addNumber(int num1, int num2)
 {
  return(num1+num2);
 }
@GetMapping("/subtract")
public int subtractNumber(int num1, int num2)
 {
  return(num1-num2);
 }
@GetMapping("/multiply")
public int multiplyNumber(int num1, int num2)
 {
  return(num1*num2);
 }
 @GetMapping("/divide")
public float divideNumber(float num1, float num2)
 {
  return(float)(num1/num2);
 }
}
```

This is a completely working service, but this does not tell users how to use it. How do we call a service? How many arguments will it take? Does it take decimals?

So without proper documentation, a service is not in a usable state. Anyone trying to use this service would be confused about what the service does and how to use it, so we need to make sure we provide proper documentation to fellow developers and end users.

Let's start with the easiest option, that is, Swagger.

Swagger

Swagger is an auto-documentation-generation tool. It is simple to use.

To start with, we need to add maven dependencies, as follows:

```
<dependency>
<groupId>io.springfox</groupId>
<artifactId>springfox-swagger-ui</artifactId>
<version>2.6.1</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>io.springfox</groupId>
<artifactId>springfox-swagger2</artifactId>
<version>2.6.1</version>
<scope>compile</scope>
</dependency>
```

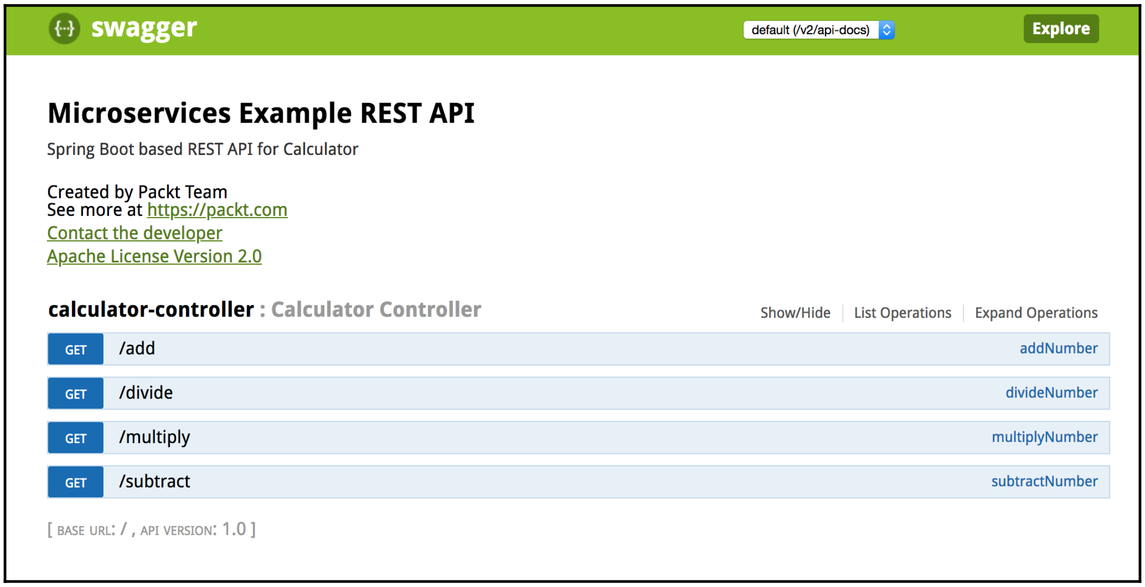
And next, we will add the configuration:

```
package com.packt.Microservices.calculator;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;
import static springfox.documentation.builders.PathSelectors.regex;
```

```
/**** This is main configuration class for spring boot application.*/
@Configuration
@SpringBootApplication
@EnableSwagger2
public class CalculatorApplication
{
 /**** Swagger configuration**/

 @return
 @Bean
 public Docket SwaggerDocumentationApi()
 {
   return new
Docket(DocumentationType.SWAGGER_2).select().apis(RequestHandlerSelectors.b
asePackage("com.packt.Microservices.calculator")).paths(reg
ex("/.*")).build().apiInfo(metaData());
 }

 private ApiInfo metaData()
 {
  ApiInfo apiInfo = new ApiInfo("Microservices Example REST  API","Spring
Boot based REST API for Calculator","1.0","Terms of service- NA",new
Contact("Packt Team", "https://packt.com", "admin@packt.com"),"Apache
License Version 2.0","https://www.apache.org/licenses/LICENSE-2.0");

  return apiInfo;
 }

 /**** Main method to initialize spring boot application.* @param  args*/
 public static void main(String[] args)
 {
  SpringApplication.run(CalculatorApplication.class, args);
 }
}
```

You can see that all we have done is added a configuration for Swagger, and no code changes are done.

After the application is deployed locally, if we open the Swagger link, `http://localhost:8080/swagger-ui.html#/`, we can see the documentation shown by Swagger. The following image shows a sample documentation:



Microservices Example REST API

Spring Boot based REST API for Calculator

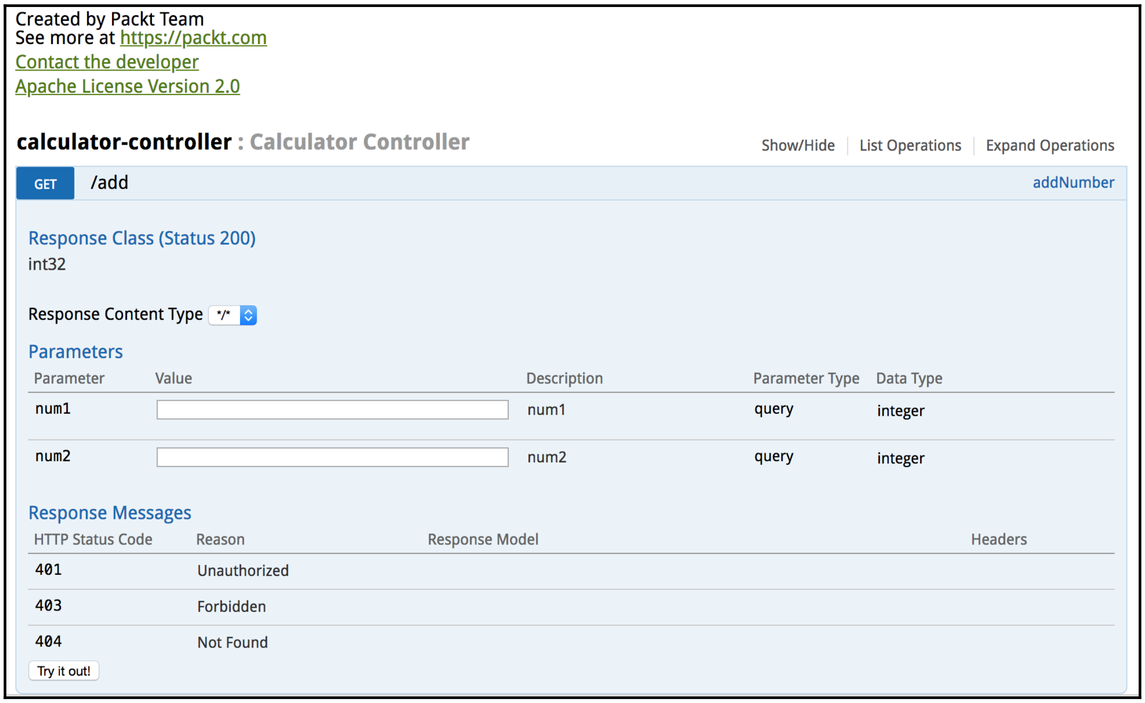
Created by Packt Team
See more at https://packt.com
Contact the developer
Apache License Version 2.0

calculator-controller : Calculator Controller

GET	/add	addNumber
GET	/divide	divideNumber
GET	/multiply	multiplyNumber
GET	/subtract	subtractNumber

[BASE URL: / , API VERSION: 1.0]

Swagger gets us to the next level; that is, click on any service and Swagger will give you more details about the service. For example, if we click on add service, we get to see something like the following on screen:



Created by Packt Team
See more at https://packt.com
Contact the developer
Apache License Version 2.0

calculator-controller : Calculator Controller

/add addNumber

Response Class (Status 200)
int32

Response Content Type */*

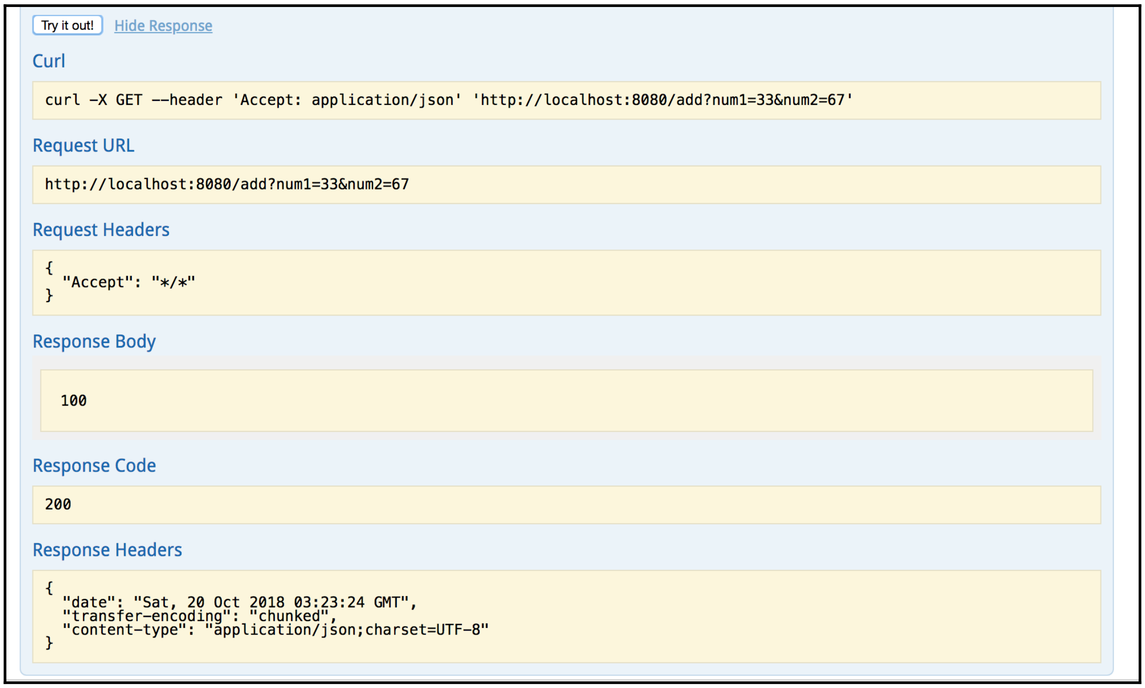
Parameters

Parameter	Value	Description	Parameter Type	Data Type
num1		num1	query	integer
num2		num2	query	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

When you click on a particular service, Swagger gives us complete details about the service. In addition, to just view the details, you can also try out the service in the Swagger interface itself. The image below shows how to use the service:



Curl

```
curl -X GET --header 'Accept: application/json' 'http://localhost:8080/add?num1=33&num2=67'
```

Request URL

```
http://localhost:8080/add?num1=33&num2=67
```

Request Headers

```
{
  "Accept": "*/*"
}
```

Response Body

```
100
```

Response Code

```
200
```

Response Headers

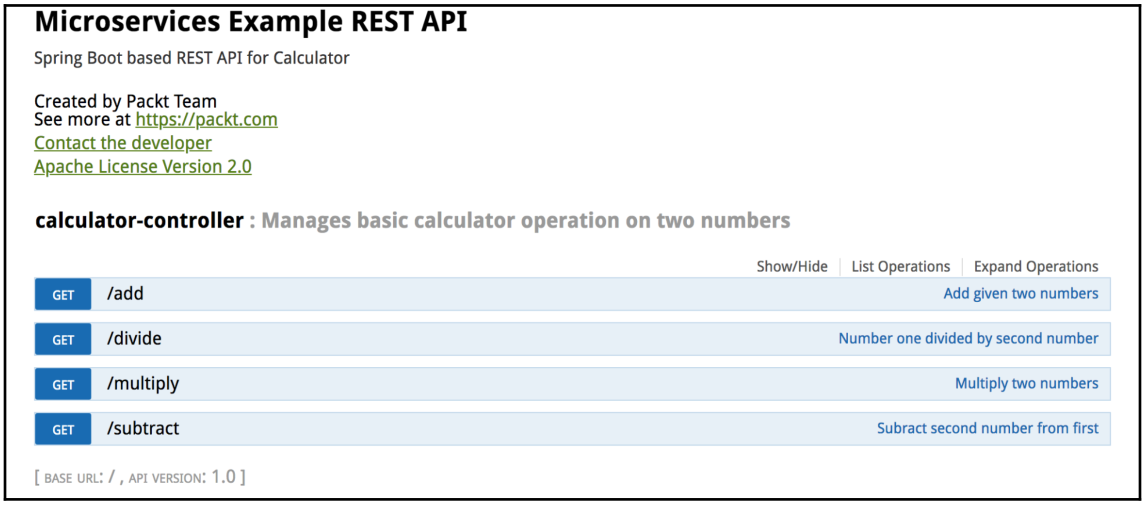
```
{
  "date": "Sat, 20 Oct 2018 03:23:24 GMT",
  "transfer-encoding": "chunked",
  "content-type": "application/json;charset=UTF-8"
}
```

So far, we have looked at the default documentation provided by Swagger, but it also gives us additional options where we can add more details to our documentation. Let's take a look at a few available options.

You can use the `@Api` and `@Apioperations` annotations, as shown in the following code, to add information at the API class and operation levels, respectively:

```
@Api(value="SimpleCalculator", description="Manages basic calculator
operation on two numbers")
@RestController
public class CalculatorController
{

  @ApiOperation(value = "Add given two numbers", response = Integer.class)
  @GetMapping("/add")
  public int addNumber(int num1, int num2)
  {
    return(num1+num2);
  }

  @ApiOperation(value = "Subract second number from first", response =
Integer.class)
  @GetMapping("/subtract")
  public int subtractNumber(int num1, int num2)
  {
    return(num1-num2);
  }

  @ApiOperation(value = "Multiply two numbers", response = Integer.class)
  @GetMapping("/multiply")
  public int multiplyNumber(int num1, int num2)
  {
    return(num1*num2);
  }

  @ApiOperation(value = "Number one divided by second number",  response =
Float.class)
  @GetMapping("/divide")
  public float divideNumber(float num1, float num2)
  {
    return(float)(num1/num2);
  }
 }
```

This will add additional information to the documentation, as shown here:


Microservices Example REST API

Spring Boot based REST API for Calculator

Created by Packt Team
See more at https://packt.com
Contact the developer
Apache License Version 2.0

calculator-controller : Manages basic calculator operation on two numbers

Show/Hide | List Operations | Expand Operations

GET /add Add given two numbers
GET /divide Number one divided by second number
GET /multiply Multiply two numbers
GET /subtract Subract second number from first

[BASE URL: / , API VERSION: 1.0]

Additionally, we can provide more information related to the service response, as shown in the following code:

```
@ApiResponses(value = {
@ApiResponse(code = 200, message = "Successfully Added two numbers."),
@ApiResponse(code = 401, message = "You are not authorized to view the
resource"),
@ApiResponse(code = 403, message = "Accessing the resource you were trying
to reach is forbidden"),
@ApiResponse(code = 404, message = "The resource you were trying to reach
is not found")
})
```

Most of the time, the default Swagger documentation will suffice, but as mentioned above, we can enhance the documentation by adding more details through annotations.

While we are talking about annotations, there is another simple way to document your APIs – through a tool called **APIdoc**, which provides us simple annotations that can help us with to document our Microservice APIs. Let's take a look at this in the next section.

APIdoc

We have talked about Swagger, which is more of an automated way of generating documentation and is useful in most cases, but there are times when you would like to have more control over what information you document and share with users. APIdoc (`http://apidocjs.com/`) is a very simple tool that helps us generate the documentation based on annotations. It gives us a lot of control, and it is similar to JavaDocs, which developers are already used to.

To get started, install APIdoc using the following command:

```
sudo npm install apidoc –g
```

Once APIdoc is installed, we are provided with a set of parameters that can be used to define documentation.

The following code showcases use of APIdoc:

```
package com.packt.Microservices.calculator2;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
/***** This class implements Calculator functions***/
@author packt
```

```
@RestController
public class CalculatorController
{

 /***Request to add 2 numbers*/
 @api {get} /add?num1={num1}&num2={num2}
 * @apiName Add
 * @apiGroup Calculate
 * @apiVersion 1.0.0
 ** @apiParam {Number} num1 first number
 * @apiParam {Number} num2 second number
 ** @apiSuccess {Number}
 @GetMapping("/add")
 public int addNumber(int num1, int num2)
 {
   return(num1+num2);
 }

 /***Request to subtract 2 numbers*/
 @api {get} /subtract?num1={num1}&num2={num2}
 * @apiName Subtract
 * @apiGroup Calculate
 * @apiVersion 1.0.0
 ** @apiParam {Number} num1 first number
 * @apiParam {Number} num2 second number
 ** @apiSuccess {Number}
 @GetMapping("/subtract")
 public int subtractNumber(int num1, int num2)
 {
   return(num1-num2);
 }

 /***Request to multiply 2 numbers*/
 @api {get} /multiply?num1={num1}&num2={num2}
 * @apiName Multiply
 * @apiGroup Calculate
 * @apiVersion 1.0.0
 ** @apiParam {Number} num1 first number
 * @apiParam {Number} num2 second number
 ** @apiSuccess {Number}
 @GetMapping("/multiply")
 public int multiplyNumber(int num1, int num2)
 {
   return(num1*num2);
 }

 /***Request to divide 2 numbers*/
 @api {get} /divide?num1={num1}&num2={num2}
```

```
    * @apiName Divide
    * @apiGroup Calculate
    * @apiVersion 1.0.0
    ** @apiParam {Number} num1 first number
    * @apiParam {Number} num2 second numbe
    ** @apiSuccess {Number}
    @GetMapping("/divide")
    public float divideNumber(float num1, float num2)
    {
      return(float)(num1/num2);
    }
   }
```

Finally, add `apidoc.json` with basic properties, as follows:

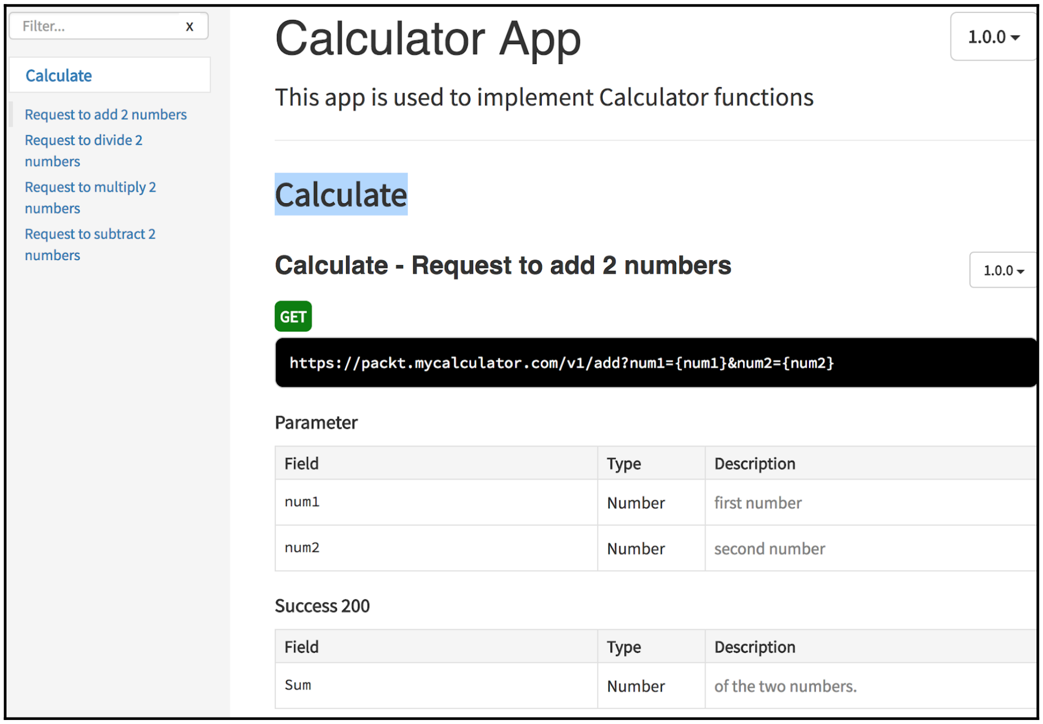
```
{
 "name": "Calculator App",
 "version": "1.0.0",
 "description": "This app is used to implement Calculator functions",
 "title": "Calculator",
 "url" : "https://packt.mycalculator.com/v1"
}
```

Now, we just need to generate the documentation, as follows:

```
apidoc -i /Users/kamalmeetsingh/Downloads/calculator2/ -o
/Users/kamalmeetsingh/Downloads/calculator2/docs/
```

This will generate the documentation in the output directory. We can open `index.html` to view the documentation. We can deploy the documentation directory on a server to make it publicly available.

An image of the output is as follow:


Filter...
x
Calculate
Request to add 2 numbers
Request to divide 2 numbers
Request to multiply 2 numbers
Request to subtract 2 numbers
1.0.0
GET

Calculator App

This app is used to implement Calculator functions

Calculate

Calculate - Request to add 2 numbers

```
https://packt.mycalculator.com/v1/add?num1={num1}&num2={num2}
```

Parameter

Field	Type	Description
num1	Number	first number
num2	Number	second number

Success 200

Field	Type	Description
Sum	Number	of the two numbers.

This kind of documentation gives us complete control over the documentation that we are creating. APIdoc gives us some core attributes, which can be used to generate documentation.

You can see how we have used some attributes in the following:

- `@api`: Signature of API
- `@apiName`: Name of API
- `@apiGroup`: Group-related APIs
- `@apiVersion`: Versions of API
- `@apiParam`: Define parameters
- `@apiSuccess`: Response on success

Apart from the preceding ones, there are a few more important attributes to mention, which you might want to use as and when needed:

- `@apiError`: Error return parameter.
- `@apiErrorExample`: Example of an error return message, output as a preformatted code.
- `@apiHeader`: Describe a parameter passed to your API-Header, such as for authorization.
- `@apiHeaderExample`: Example showcasing the headers.
- `@apiSampleRequest`: Example showcasing the sample request.

You can learn more about APIdocs at `http://apidocjs.com/`. This can be a very powerful tool for API documentation, which allows us to provide as much detail as we would like.

Additional Documentation Frameworks

So far, we have looked at the Swagger and APIdoc libraries. There are many additional tools that can help us in documenting our services and Microservices. In the end, it will come down to the preference of the development team. It is worth talking about Enunciate and the Spring REST docs before we move ahead to the next topic:

- **Enunciate**: This is an open source project, licensed under Apache License. It is easy to integrate with existing web projects and has a set of annotations available to support documentation. More information can be found at `http://enunciate.webcohesion.com/`,`https://github.com/stoicflame/enunciate/wiki`, which has a sample project code as well.
- **Spring REST docs**: Will it be beneficial if we can create our documentation while we are implementing our test cases? That is the thought behind Spring REST docs. Though it does add some dependency on unit testing, it is like killing two birds with one stone. If you are already using the Spring framework, you should definitely give the Spring REST docs a try. For more details and some sample usage, check out `https://spring.io/projects/spring-restdocs`.

Testing Microservices

Testing is an important factor for any application. You want to make sure that your application and Microservices will work fine and withstand cases when things go wrong. We can look at testing at different levels, such as unit testing, integration testing, service-level testing, and end-to-end testing.

We will cover each of these aspects in detail.

Unit testing

The idea is to test each unit of your application independently. This is mostly easy to implement, as we are focusing on only one unit at a time. For each unit, we need to think about all the cases that can occur. For example, let's take the simple division method from our previous example, as follows:

```
public float divideNumber(float num1, float num2)
{
  return(float)(num1/num2);
}
```

We will need to think whether the divide by zero error is handled. Can I divide negative numbers? Are fractions being handled? For a simple single-line method, we can think of so many scenarios. So, let's take a look at what our Unit-testing class would look like:

```
package com.packt.Microservices.calculator2;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
@RunWith(SpringRunner.class)
@SpringBootTest
public class Calculator2ApplicationTests
{
  /*** This method tests division of 2 integers*/
  @Test
  public void divideIntegerSuccessTest()
  {
    CalculatorController calculatorController = new
CalculatorController();
    int num1 = 10;
    int num2 = 5;
    float expectedResult = 2;
    float actualResult = calculatorController.divideNumber(num1, num2);
    assertEquals(expectedResult, actualResult, 0);
  }
  /*** This method tests for division of two float numbers*/
  @Test
  public void divideFloatSuccessTest()
  {
     float num1 = (float) 55.5;
     float num2 = (float) 11.1;
```

```
        float expectedResult = 5;
        float actualResult = calculatorController.divideNumber(num1, num2);
        assertEquals(expectedResult, actualResult, 0);
     }
     /*** This method tests for division by negative numbers*/
     @Test
     public void divideFloatNegativeSuccessTest()
     {
       CalculatorController calculatorController = new
   CalculatorController();
       float num1 = (float) -55.5;
       float num2 = (float) -11.1;
       float expectedResult = 5;
       float actualResult = calculatorController.divideNumber(num1, num2);
       assertEquals(expectedResult, actualResult, 0);
     }
     /*** This method tests for division by negative numbers*/
     @Test
     public void divideByZeroTest()
     {
       CalculatorController calculatorController = new CalculatorController();
       int num1 = 30;
       int num2 = 0;
       float expectedResult = Float.POSITIVE_INFINITY;
       float actualResult = calculatorController.divideNumber(num1, num2);
       assertEquals(expectedResult, actualResult, 0);
     }
    }
```

We have written so many test cases for a single line of code. One important question that comes up time and time again is, "How much unit testing is sufficient?" One way to track that is through code coverage. We should normally target for 90% code coverage. But this number can go up or down based on your application. For example, in our case, we have only a single line of code, so we will definitely achieve 100% coverage with just a single test case. But it is good to cover just as many corner cases as well, for example, if an error occurs at some point, how the application would behave.

When talking about Unit testing, another important factor is mocking. Let's say your code interacts with additional code, which might be a part of the same service or a different service. We would ideally like to mock any such outsider service, as all we want to test right now is a current unit in hand, that is the unit we want to validate.

To understand the power of mocking, let's extend our previous example a little bit. Let's say we need to divide a number by a constant value, and this value will be supplied by another third-party service. We will use the constructor injection to add the third-party dependency, as follows:

```
FetchConstantService fetchConstantService = new FetchConstantService();
CalculatorController(FetchConstantService fetchConstantService)
{
  this.fetchConstantService = fetchConstantService;
}
/***Request to divide 2 numbers*/
@api {get} /dividebyconstant?num1={num1}&num2={num2}
* @apiName Divide
* @apiGroup Calculate
* @apiVersion 1.0.0
** @apiParam {Number} num1 first number
** @apiSuccess {Number} Quotient when num1 is divided by constant.
 @GetMapping("/dividebyconstant")
public float divideNumberByConstant(float num1)
{
  float num2 = fetchConstant();
  return(float)(num1/num2);
}
/**** This method return a constant value based on a third party service*/
* @return
public float fetchConstant()
{
  return fetchConstantService.getConstantValue();
}
```

The `divideNumberByConstant` method is of interest to us. We would like to test how this method would work. But we can clearly see that this method calls another method, and that another method calls a third-party service. Now we do not have any control over the third-party services; moreover, we might not want to test them, as they would be tested by the team providing them. So, why waste time testing something that is already tested! In such cases, it is logical to mock any dependencies that we don't want to test.

There are many libraries to implement mocking. As an example, we will use Mockito, as follows:

```
/*** This method tests divide by constant*/
@Test
public void divideByConstantSuccessTest()
{
  FetchConstantService fetchConstantService =
Mockito.mock(FetchConstantService.class);
  when(fetchConstantService.getConstantValue()).thenReturn((float) 2);
```

```
    CalculatorController calculatorController = new
  CalculatorController(fetchConstantService);
    int num1 = 10;
    // define return value for method
    getUniqueId()
    float expectedResult = 5;
    float actualResult =  calculatorController.divideNumberByConstant(num1);
    assertEquals(expectedResult, actualResult, 0);
  }
```

The idea is to assume that, when the third-party service works fine, our code should work well.

Integration Testing

In the last section, we talked about unit testing, where we tested each individual unit independently and recommended mocking any external services. With integration testing, our focus is the reverse, that is, we actually want to test whether our service is able to communicate with any outside resources or services. If the other service is down or not reachable, will our service handle the scenario gracefully?

So, revisiting our example, where we talked about fetching a constant value from a third-party service, we have the following:

```
FetchConstantService fetchConstantService = new FetchConstantService();
CalculatorController(FetchConstantService fetchConstantService)
{
  this.fetchConstantService = fetchConstantService;
}

/**Request to divide 2 numbers*/
* @api {get} /dividebyconstant?num1={num1}&num2={num2}
* @apiName Divide
* @apiGroup Calculate
* @apiVersion 1.0.0
** @apiParam {Number} num1 first number
** @apiSuccess {Number}
@GetMapping("/dividebyconstant")
public float divideNumberByConstant(float num1)
{
  float num2 = fetchConstant();
  return(float)(num1/num2);
}
/**** This method return a constant value based on a third party service
* @return*/
```

```
public float fetchConstant()
{
  return fetchConstantService.getConstantValue();
}
```

We are using a service that is a black box to us. We do not know what is happening behind the scenes; it might be fetching the data from some file, some database, some third-party service, or so on. This black box can fail or behave in an unanticipated manner. That is what integration testing is all about! We will not mock `fetchConstant` this time, and see what will happen in the following code:

```
/*** This method tests divide by constant- no mocks*/
@Test
public void divideByConstantSuccessNoMockTest()
{
  FetchConstantService fetchConstantService = new  FetchConstantService();
  CalculatorController calculatorController = new
CalculatorController(fetchConstantService);
  int num1 = 10;
  // define return value for method
  getUniqueId()
  float expectedResult = 2;
  float actualResult =  calculatorController.divideNumberByConstant(num1);
  assertEquals(expectedResult, actualResult, 0);
}
```

This test case actually brings out a lot of flaws in our code. For example, we have not handled the cases where the black box service is not responding or taking a lot of time. When we start integration testing, we will be forced to harden our service, to make sure we handle all the corner cases.

The following code shows an example of an integration test:

```
@GetMapping("/dividebyconstant")
public float divideNumberByConstant(float num1)
{
  float num2;
  try
  {
    num2 = fetchConstant();
  }
  catch(Exception e)
  {
    // In real code we will handle multiple exceptions expected
    // and not just the high level exception
    // Once an error occur, we will need to handle it.
    // We will need to log the proper error and handle it gracefully
```

```
        // so that or code does not break.
        num2 = 2;
      }
      return(float)(num1/num2);
    }
```

We also need to add timeouts to make sure our services do not get stuck for long periods of time. These best practices help our code to be robust and disaster-ready. A good integration test brings out loopholes in your code and fixes the problems before end users start reporting them.

Service Testing

Here, we are talking about testing our Microservice as a whole. For example, in the calculator service example we used throughout the chapter, we talked about making sure all the methods in the service are tested. Well, this is a very simple service, but in most cases, it would mean that all the layers (data, business, and so on) in a service work fine. A more real-world example would be that we are updating data for an employee and we hit a service with some data, which in turn calls a business validation layer to validate the correctness of the data. After validation, a data layer would be called and, finally, the data is stored. A service-level test would mean testing all these layers. We will discuss these layers in detail in the *Levels of testing* section.

Testing a Microservice from end to end can be complicated, as at times it might mean we need to bring up the service, make calls to the endpoint, and make sure things work fine. There are many tools available for our help. Here, I will use MockMVC, which integrates well with Spring Boot. This will demonstrate how we can test an end-to-end service. We will use the previous example for the calculator service, specifically the add operation, as follows:

```
@GetMapping("/add")
public int addNumber(int num1, int num2)
{
  return(num1+num2);
}
```

Here is the code to test the preceding service:

```
//We will create a simple test to do end to end testing
package com.packt.Microservices.calculator2;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;
@RunWith(SpringRunner.class)
@WebMvcTest(CalculatorController.class)
public class RestTest
{
  @Autowired
  private MockMvc mvc;
  @Test
  public void testRESTAdd()
  throws Exception
  {
     mvc.perform( MockMvcRequestBuilders.get("/add?
num1=1&num2=2")).andExpect(MockMvcResultMatchers.status().isOk()).andExpect
(MockMvcResultMatchers.content( ).string("3"));
  }
}
```

The code that we are interested to test is the way we made a call to our add service. We make a mock call to the service and validated the end result, that is, we sent `1` and `2` as an input to the service and validated `3` as an output of the add operation.

End-to-end testing

Our application is not a standalone Microservice but a combination of multiple Microservices, which are supposed to work together to make sure the end user gets the results for which the application is created. With the end-to-end test, we try to think from the end user's perspective and make sure the application works as a whole. This is the most tricky and time-consuming test.

In a web application, the end-to-end testing would mean that all features, services, database, queues, and external dependencies in the application work fine. So, if we have a simple form to be submitted through an interface, and we have a Microservice or a bunch of Microservices to receive, validate, and manipulate the data provided, end-to-end testing would mean making sure that everything is working fine as a whole system.

At times, we will focus on negative testing. Say one of many services is down, how will it impact our application as a whole? Will our application handle such a scenario gracefully, or will it bring our whole application down?

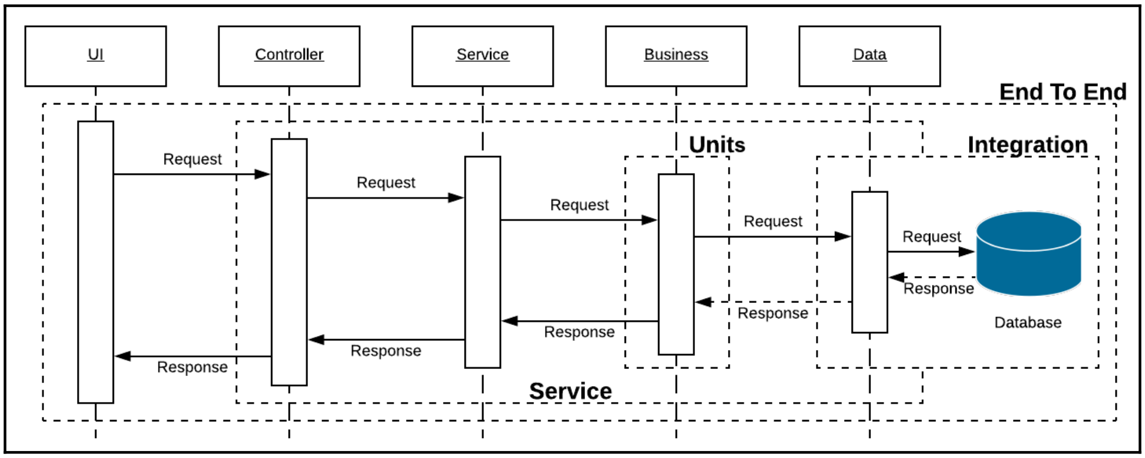
At times, we will rely upon manual testing to make sure our application works from end to end. But again, there are tools that help us test an application from end to end. A popular one is Selenium (`https://www.seleniumhq.org`). Tools such as Selenium help us automate the browser testing, that is, we can define the user flow right from entering the URL into the browser, fill in the forms, click the submit buttons, and validate resultant responses. Selenium has gained a lot of popularity recently because of it's open source and has strong community support. You can also go for licensed solutions, such as **Unified Functional Testing** (**which used to be popularly known as QTP**).

The bottom line is that you can go for an open source or licensed solution, or even for manual end-to-end testing, but it is very important to make sure that we look at the solution from an end user's perspective.

Levels of Testing

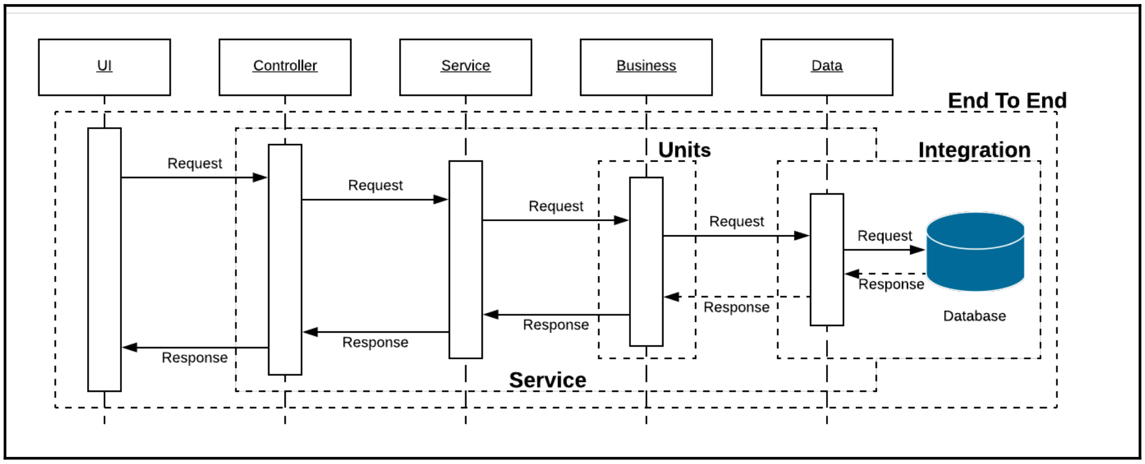
We talked about different levels of testing that can be done when we create an application that uses Microservices. Let's take an example to clarify the difference between different types of testing and how useful each is. Say we have a form that lets the user enter employee details that get saved to a database and a success message is returned to the calling client. Additionally, say we use a layered design, with **Controller**, **Service**, **Business,** and **Data**.

The following diagram should make the design clear:



- **Unit Testing**: Each class and each method is a unit in itself and can be tested independently. For example, say we take a look at the business layer, which might have many classes. We will take each class, and then try to test each method. In an ideal world, we would test each unit independently, but this might be time-consuming. So, based on time availability, it might make sense to understand core areas of your application and implement unit testing for those.
- **Integration Testing**: Most of the time, our services cannot work in isolation. In the preceding example, our service is actually talking to a database. We need to test whether our service will work when the database is available and able to handle cases when the database is not responding or is slow.
- **Service Testing**: Here we will focus on the service as a whole. We might want to mock any external dependencies, but we will make sure all the code that is part of our service – all the layers – should be tested.
- **End-to-end Testing**: This includes testing from the first layer – the UI – until the last layer – the database.

Putting it all together, the following diagram should give you a clearer picture about what different levels of testing indicate:



The preceding markings should help you visualize the different layers of testing.

Summary

In this chapter, we discussed two important aspects of Microservices: documentation and testing. Documentation is important for any software so that others can understand how to use the services you have created. Without proper documentation, it will be hard for anyone to understand how to use the services properly, what types of input are expected, what outputs are returned, and what kind of error messages to anticipate. We talked about Swagger and APIdocs for documentation and touched upon a few additional frameworks that can help.

We talked about the various levels of testing that can be done for a Microservices-based architecture. In unit testing, we would try to test each class and method as independent units and make sure each smallest unit works fine independently, by mocking any unnecessary dependencies. While implementing integration testing, we would make sure that our service is able to interact with external dependencies and handle the problem scenarios gracefully. Service-level testing is responsible for testing the complete Microservice. Finally, end-to-end testing makes sure the system will work fine as a whole for the end user.

Index