Umesh Ram Sharma

Practical Microservices

A practical approach to understanding microservices

Packt>

Practical Microservices

A practical approach to understanding microservices

Umesh Ram Sharma

BIRMINGHAM - MUMBAI

Practical Microservices

First published: July 2017

Production reference: 1270717

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

ISBN 978-1-78588-508-2

www.packtpub.com

Credits

Author
Umesh Ram Sharma

Reviewer
Yogendra Sharma

Acquisition Editor
Chaitanya Nair

Content Development Editor
Zeeyan Pinheiro

Technical Editor.
Vivek Pala

Copy Editor
Karuna Narayan

Project Coordinator
Vaidehi Sawant

Proofreader
Safis Editing

Indexer
Francy Puthiry

Production Coordinator
Nilesh Mohite

About the Author

Umesh Ram Sharma is a developer with more than 8 years of experience in the architecture, design, and development of scalable and distributed cloud-based applications.

He was awarded a master's degree from Karnataka State Open University in Information Technology. With a core interest in microservices and Spring, he is an expert in the utilization of various offerings of the J2EE, Java Script, Struts, Hibernate, and Spring stack. He also has hands-on experience of technologies such as AWS, J2EE, MySql, MongoDB, memchached, Apache, Tomcat, and Hazelcast.

Currently working as a Principal Lead Engineer at ZestMoney, he helps his team migrate their running project to microservices.
In his free time, he enjoys driving, cooking, and attending conferences on new technologies.

About the Reviewer

Yogendra Sharma is a Java developer with a Python background, with experience mainly in middleware development. He has completed a bachelor's degree of technology in computer science.

He is currently working in Pune at Intelizign Engineering Services Pvt. Ltd as an IoT Cloud Architect. He constantly explores technical novelties, and he is open-minded and eager to learn about new technologies and frameworks.

Yogendra has also technically reviewed *Mastering Python Design Patterns*, *Test-Driven Development with Django*, *Spring 4.0 Microservices*, and *Distributed Computing with Java 9*, and video courses *Python Projects*, *Learning Python Data Analysis*, *Django Projects: E-Learning Portal*, and *Basic and Low-Level Python Network Attacks*, all by Packt.

I would like to thank my parents for allowing me to learn all that I did. I would also like to thank my friends for their support and encouragement.

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/1785885081`.

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 is becoming a buzzword in the technology world. It is getting lots of attention among technology enthusiasts, developers, and articles. This book is intended to be a practical guide for developers to write, test, secure, and deploy microservices. It has many benefits and also unique challenges; I intend to shed light on the best practices in an easy-to-read manner, and also to avoid the pitfalls associated with the complexity that can arise due to this architecture.

What this book covers

`Chapter 1`, *Introduction to Microservices Architecture*, introduces the overall concepts of a microservices architecture, including a basic definition of what is meant by microservices in this book. The chapter will then quickly move to a high-level walk-through of a comprehensive sample application that will be used throughout the rest of this book.

`Chapter 2`, *Defining Microservice Components*, explains the guiding principles to define microservice components, along with teaching how these components form the backbone of the microservices architecture. These guidelines are then made practical by showing how a Spring Boot-based Java project structure can be used to effectively define microservice components. Finally, a sample microservice is used to demonstrate a practical example of Java-based microservice components, along with configuration and a discovery service.

`Chapter 3`, *Communication between Microservice Endpoints*, talks about the principles for effective communication between microservices, along with the appropriate reasoning. Options are then introduced for synchronous and asynchronous communication using various technologies ranging from Spring Framework's own capabilities to message brokers. We will also cover how common problems are handled by best practices.

`Chapter 4`, *Securing Microservice Endpoints*, discusses common security and security challenges. JWT, OpenID, and OAuth2.0 are introduced for better security in a microservices architecture.

`Chapter 5`, *Creating an Effective Data Model*, starts with explaining the difference between creating a microservices-based data model versus a traditional data model and why microservices does it differently. It also explains how data technologies can be intermixed, and how to select the appropriate data management strategy for each microservice component. We will also walk through the sample application data model and explain the various data model decisions and why they were made.

`Chapter 6`, *Testing Microservices*, talks about how testing is even more important in a system that is intended to be changed constantly and deployed automatically. However, are our traditional testing approaches and test quality criteria a perfect match for a microservice-style architecture, or do we need to focus on entirely different approaches? Probably a mix of both.

`Chapter 7`, *Deploying Microservices*, lets us know that in microservices architecture, we will have a lot to deploy and fairly often. That's one reason to make deployment as painless and easy as possible. Another reason is the fact that we strive for automation and expect our system to be easily scaled up and down, meaning that new microservices are deployed and killed constantly. Docker will help us define and automate the deployment process of microservices.

`Chapter 8`, *Evolving the Existing System*, describes the primary mechanisms to evolve a system built on a microservices architecture and how microservices enable such evolution. This chapter will also walk us through an example of how to evolve the sample Java-based application.

`Chapter 9`, *Monitoring and Scaling*, describes the key concepts and principles in monitoring and scaling a microservices-based system. This chapter explores the practical approaches to monitoring and scaling Java-based microservices and gives us examples of how to monitor and scale the sample application.

`Chapter 10`, *Troubleshooting*, reviews common issues encountered when designing and building a microservices-based architecture and describes common approaches to solve or mitigate them.

What you need for this book

To run examples given in this book, it is recommended that you have a Linux-flavored operating system. You can use Windows or Mac-based systems as well, but the steps described here to install any particular software are explained, keeping the Linux-based system in mind. Other than that, you need to have Maven, Java 8, and any Java-based IDE. It can be Eclipse, IntelliJ, or STS. For the database side, the MySQL community version is recommended.

Who this book is for

This book is for Java developers who want to get started with microservices and implement it in their work place. No knowledge of microservices is necessary.

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, path names, dummy URLs, user input, and Twitter handles are shown as follows: "In the `/src/main/java/com/sample/firstboot/controller` folder, create a called `SampleController.java` file."

A block of code is set as follows:

```
@SpringBootApplication
@EnableZuulProxy
public class ApiGatewayExampleInSpring
{
    public static void main(String[] args)
    {
        SpringApplication.run(ApiGatewayExampleInSpring.class, args);
    }
}
```

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

```
curl http://localhost:8888/userService/default
```

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: "If any user wants to set more specific details regarding the project, then they can see all the configuration settings by clicking on the **Switch to the full version** button."

Warnings or important notes appear 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 for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `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/Practical-Microservices`. 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.

1

Introduction to Microservices Architecture

Software architecture can be defined as a set of rules or principles designed for a system that defines the elements, behavior, structure, and relationship between different components of a software system.

In the early 80s, when large-scale software systems started coming into existence, a need emerged to have generic patterns (or architecture) that would solve common problems faced while designing all those systems. The term ,"software architecture" started evolving from there. Since then, many types of architectures have been introduced to design large-scale software systems. The software industry has seen architecture styles from **shared nothing** to **monolithic** to **client-server** to **n-tire** to **service-oriented architecture** (**SOA**) and many more. One more addition to this list is **microservice architecture**.

Microservice is a word that has taken an exponential path of popularity in recent times among the software developer/architecture community. Organizations working on monolithic application architecture often complain about long release cycles, tedious debugging issues, high-maintenance road maps, scaling issues, and so on. The list is never ending. Even the exceptionally well-managed monolithic applications would need a tremendous amount of effort to handle these issues. Microservice is evolving rapidly as it offers an efficient way to cope with these kinds of issues. In simple words, we can explain it as breaking a large problem into relatively smaller services, and each service will play its own part.

The basic philosophy of the microservices architecture is, *"Do one thing and do it exceptionally well*".

The heart of microservices architecture is the **Single Responsibility Principle** (**SRP**). In microservices architecture, business tasks are broken into smaller tasks, and to accomplish each task, there is a microservice. In a system, there could be two microservices or 100 microservices, depending on business requirements and how well the task has been broken down. This type of architecture gives an organization many benefits, which were not possible in monolithic applications, and also, it has its own kind of overhead as well. We will discuss the benefits and tradeoffs of this kind of architecture in the upcoming sections.

General microservices architecture

Microservice architecture is basically inspired by SOA. There is no golden rule for the architecture in microservice. If we go by different architectures implemented in the industry, we can see everybody has their own flavor of microservice architecture. There is no perfect or certain definition of microservice. Rather microservices architecture can be summarized by certain characteristics or principles.

Characteristics of microservice architecture

Any architecture demonstrating the following six principles or characteristics could be placed in the microservice architecture zone:

- The system should consist of two or more running units or components. These components should expose their functionality as services. Each component should serve a business purpose, and the components should be loosely coupled. Components should communicate with each other through predefined protocols, such as messaging queues, HTTP request/response models, and so on.
- Systems should be language agnostic. One component can be developed in Java, and another can be developed in .NET. The decision of choosing a technology platform for a particular service should not affect the application architecture.
- Systems should have a decentralized database. Ideally, each component or microservice should have its own database, with whom only that service is interacting. No other component or service can fetch or modify the data in that database.

- Each component in the system should be cohesive, independent, and self-deployable. It should not be dependent on any other component or resource to work or to deploy. It should have **Continuous Integration/Continuous Deployment** (**CI/CD**) in place to ship faster.
- The system should have automated testing in place. Speed is the one of the most desirable features of microservice architecture. In the cycle of build, test, and ship, if automated testing is not in place, then it cannot meet the goal.
- Any component/service's failure should be in isolation. Failure of one service should not make the whole application go down. If it fails, it should not affect other components/services. Some kind of failure roll back mechanism should be in place. That means if one service fails, it should be easy to roll it back to an older working version.

This simple and small example given below may give a better understanding and more insight into these principles.

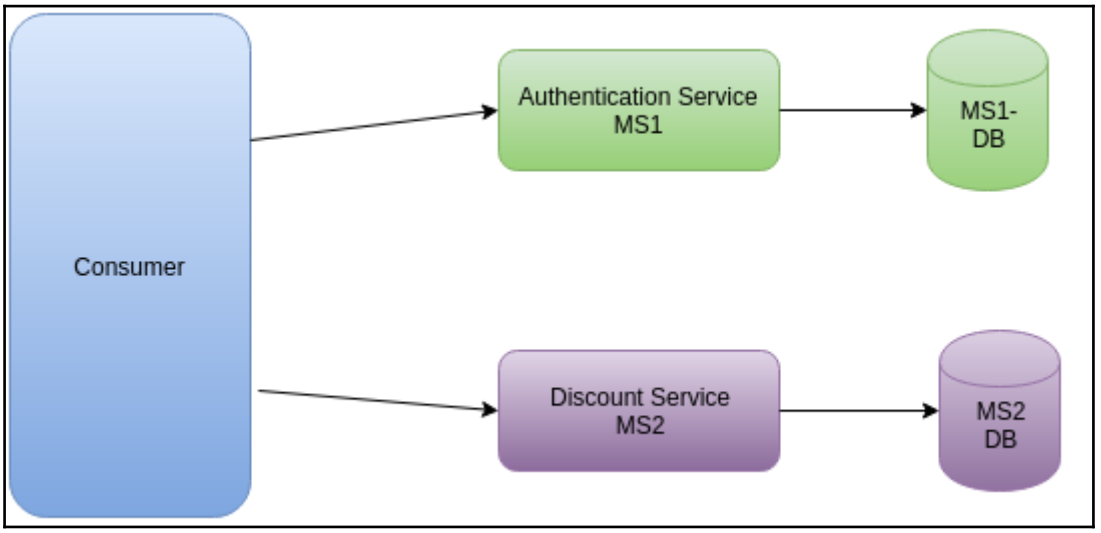
Problem definition

An application is required, that generates coupons for shopping online on the basis of privileges given to a registered user. If a user is from the platinum category, they will have a 20 percent discount coupon, gold users 15 percent, silver 10 percent, and guests 5 percent.

Solution

In this architecture, there is scope for two microservices. The first is for authentication that gets users logged in based on the credentials provided and sends the privilege level back to the consumer as a response. The second is based on privilege level. The service should return the percentage of discount that may be applied on the cart (another service) built by the consumer.

In the following diagram, there are two components having two different Databases. Assuming both components have exposed their services as REST interfaces, consumers can hit **Authentication Service MS1** with credentials and get the user object, including privilege level, and then, it can hit the second microservice with privilege level and get the discount coupon percentage:



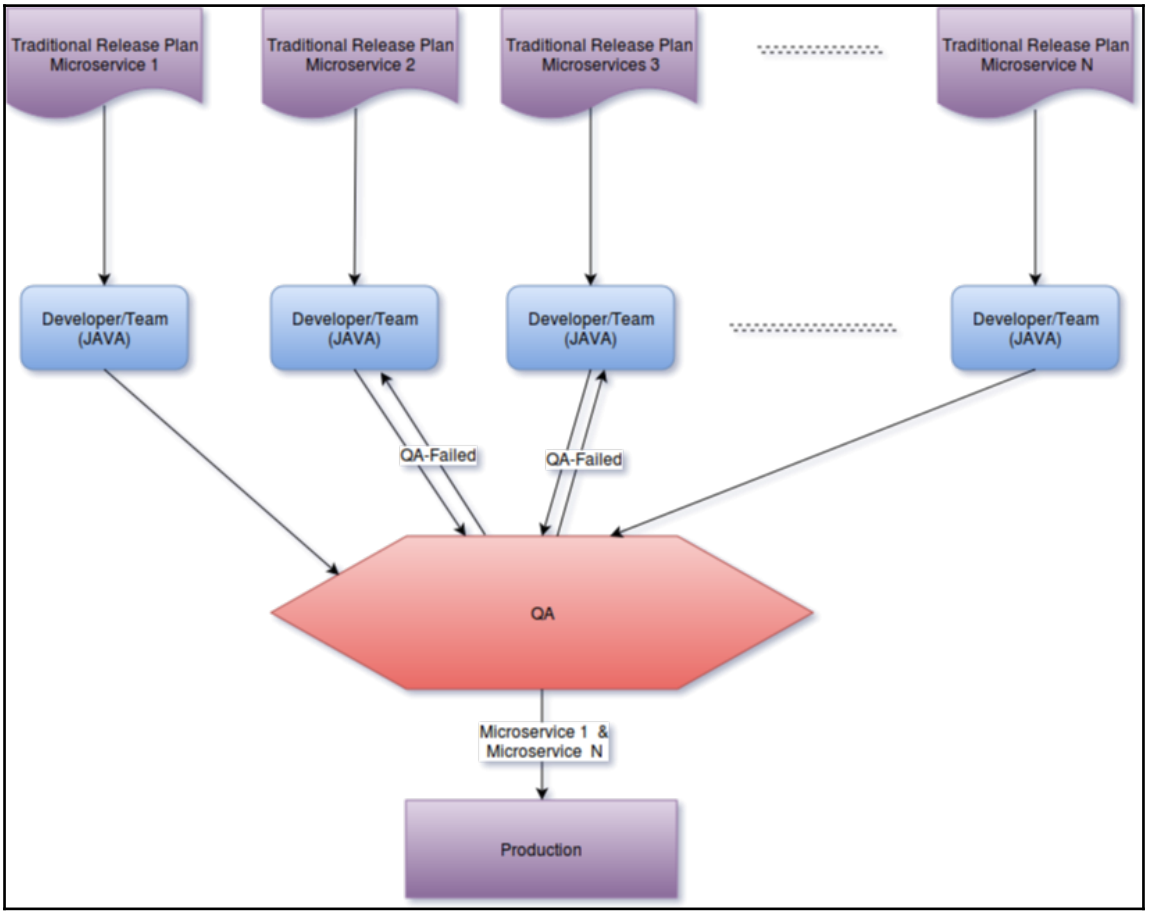
How much proposed solution is aligned with microservice architecture?

Each service servers a business purpose (rule 1). Both services can be on different platforms; it will not affect the existence of the other service (rule 2). Both services have different databases with which only that particular service is communicating (rule 3). Services are independent of each other which means not sharing any database among themselves and self-deployable (assuming CI/CD in place rule 4).

If we assume here **Authentication Service MS1** dies in some rare circumstances because of a runtime exception or memory leakage issue, the consumer will get an HTTP response of 404. 404 (Not Found) is treated as user not found in the database, which will lead the consumer to treat this user as a guest, and it will still keep asking for a discount coupon for the guest. The system/website will be up and running. Failure of **Authentication Service MS1** will not hamper any other running service. (rule 6 isolation failure.)

If the same problem occurred in a monolithic application (memory leak), it would require rebooting the whole application, including all components, and it would cause a downtime.

The success of microservice architecture is based on how efficiently one analyze the problem domain and broke down the application into smaller independent components. The better the assorting of problems, the better the architecture will evolve. The typical architecture of microservices is as shown in the following diagram:



Normally, each team is responsible for one microservice. The size of a team can vary from one member to many with the size and complexity of microservices. Each microservice has its own release plan. Deployment of one microservice doesn't affect the deployment of an other microservice. If, at a particular time, five microservices are in testing and **quality analysis** (**QA**) and only two pass the QA phase, then only those two can ship to production, and they should not be affected by or dependent on the release of the rest of the services.

Challenges for a successful microservice architecture

Microservice architecture is not a silver bullet to solve all the architecture problems in the existing world. It gives a solution to many of the problems raised, but it comes with its own challenges. Decomposing database, communication between APIs, heavy DevOps work, and a team with aligned mentality are some of the initial factors to cope with while moving towards microservices. Even with successful implementation of a microservice, here are the challenges that an organization will face.

Debugging through logging

Debugging in microservices architecture will be hard for a developer. A single request to an application can result in multiple internal hits of different microservices. Each microservice will generate its own log. To find the root cause of any wrong behavior in any particular request could be a nightmare for a developer. In a distributed logging environment, it's hard to debug any issue. Setting up proper tools/scripts to collect logs from different services in some centralized place also gives a challenge to the DevOps team.

Monitoring microservices

Monolithic applications are easy to monitor. In a monolithic application, there are fewer points to monitor as it is one service(big fat war). Points to monitor would include the database service, disk spaces, CPUs usage, any third-party tools and so on. Monitoring effort in microservices architecture increases exponentially. Each service requires the same number of points to monitor as normally found in one monolithic application. Imagine the alert rate when microservices numbers keep increasing in 100s. Monitoring and self-healing tools/script such as Nagios and Monit should be in place to handle and react to alerts coming from different services.

Common libraries

Using a shared library among different services may not be a good idea. There could be a case where one microservice A, generates user object JSON, and other microservice B, consumes user object JSON. The suggestion here is to define the user class in one JAR and add that JAR into both the microservices. This leads to two new issues. The first one is that sharing a common library will again lead microservices to stick with the same language. A secondly, this idea will increase the coupling between services, which is against the idea of microservices. Solution could be the `User` class should be written in both the microservices. This can result in the failure in the **Don't Repeat Yourself** (**DRY**) principle.

Messaging between services

In microservices architecture, serving a single request from outside (frontend or API consumer), can result in multiple internal communications between different microservices, which can result in a network latency. This can, in turn, degrade the overall application performance. It can be solved by choosing the right type of communication between APIs. It can be synchronous like RESTful web services or asynchronous like messaging. Depending on the application requirement, both types of communication can co-exist in one place. If a direct response is required, it should be synchronous like HTTP. If there is some intermediate result that should pass to the next service or services, the best way to do so is by publishing the events to topics or queues, such as Kafka or SNS.

Deployment and versioning of microservices

Microservices can be deployed in different ways. Microservices usually ship with containers such as packer, Dockers, or for **Amazon Web Services** (**AWS**), one can also use **Amazon Machine Image** (**AMI**) to ship the microservice. However, AMIs take longer to create and deploy than Dockers, and Docker is a better tool for deploying a microservice. Proper versioning rules should be in place; otherwise, the whole architecture will be in a versioning hell. Typically, `xx.xx.xx` is the versioning format. The right most is for smaller bugs or patches, while the middle section increases by adding any new feature in the component. The left most number increases when you have changed something big, like the communication interface model.

We will talk in detail about communication between services and deployment in the upcoming chapters.

Future of microservices

Along with their pros and cons, microservices are gaining popularity these days. With more and more people using this concept, there can be new problems, new patterns to implement microservices, and new demands from microservices on the table. Those demands and common problems will give the direction or lead the future of microservices.

Increasing the number of microservices in the organization also increases the DevOps work. DevOps has to deal with problems such as lots of deployment pipelines, log handling in a centralized place, monitoring service and server attributes, self-healing script, and so on. With a DevOps mindset, developers have lots to do other than only developing business logic. With an increase in DevOps effort, microservices could lead us in two directions:

- Serverless architecture
- Microservices as PaaS

Serverless architecture

Serverless architecture, with certain offerings from different cloud providers, doesn't seem too far. For instance, AWS has an incredible feature named **Lambda** function. Typically how it works is a bunch of code stays in your S3 environment as a ZIP file. Whenever a specific event is triggered, a new instance is created with predefined parameters. This particular piece of code is copied from a specified location to the new instance, and this code starts running. After finishing its job, the instance will be shut down again. With service discovery pattern, this whole concept becomes very interesting. In this type of environment, services will be coming up and down as per demand. Each new upcoming service will be registering itself and deregistering itself on going down, and it uses discovery protocol to find the service with which it needs to communicate.

The above mentioned architecture doesn't give the whole solution for serverless architecture, for instance it doesn't explain the versioning of services, how to deal with latency time, which new on demand services take to coming up for a new HTTP request and so on. The boot up time of the instance and services will be crucial in this scenario. Evolving of these type of component like Lambda from cloud providers, can lead future to server less components. Developer will be focusing on the development of business component. DevOps work will be lesser in this case.

Microservices as PaaS

Microservice as PaaS can be another future direction. Microservices will be delivered as **Platform as a service** (**PaaS**). With containers such as packer or Docker in place, this also seems to be feasible. In this scenario, microservice will be shipping with frameworks that do most of the monitoring, logging, and registering a service mechanism, common library, load balancing, versioning, and similar kind of features in itself. Containers are in place leading microservices in somewhat of the same direction. They can be seen as a framework, coming with all the monitoring, logging and so on already in themselves. Developers have to worry less about for all that stuff.

As containers become more mature and cloud components evolve, we can see the merging of both directions into one in the long run.

Dominance of microservice architecture over traditional architecture

John's story

John, a good coding developer, joins a company that deals in stock exchange software. The software provided by this company also gives users recommendation of stocks, which are likely to rise in the near future. It is based on some probability logic. John has done his homework, and the very first day, he suggested a tweak in that probability logic that can improve the prediction of stocks. His suggestion was to change a few lines in the code, and John expected to finish this before lunch. The software of the company was a monolithic application with over 40 modules inside it. It took one whole day's effort from John to clone the code, figure out in which module he has to change carry out code, and the setup to run it locally. Third-party tools stub dependencies, configurations mismatch, and many other issues took time to get figured out and be resolved to run the application.

So, challenges faced by John in a monolithic environment can be summarized, as generic problems in monolithic, in the following words:

- When we talk about monolithic, we are talking about a single large code base. It may or may not be in the small sub modules, but they are not self-dependent or self-deployable.
- This whole code base is written in a single specific language. Excited or new developers are bound to work in that technology.
- Being a single bundle application, if traffic increases, the whole application has to scale, even if that traffic is affecting only one module of the whole code base. A monolithic application can be scaled only horizontally by running multiple instances of the whole application.
- As the code base grows, it becomes more prone to bugs while introducing any new feature. As modules may be interdependent of each other, the testing cycle and deployment cycle increase in the case of a monolithic application.
- It is hard to find an ownership sense in the case of a monolithic application. Employees generally don't like to take ownership of such big projects.

In other scenario, John suggested a tweak. The application was on microservice architecture. So, John clones the receptive repository. Code cloning was fairly instant (less code, less branches). With less code, John easily figures out where to make changes. Running this microservice in a local environment was fairly easy. He committed the code and pushed for review. CI/CD is in place, so his code merged to branch and deployed in, say DevOps, the Dev environment. John was ready to ship the project just after lunch.

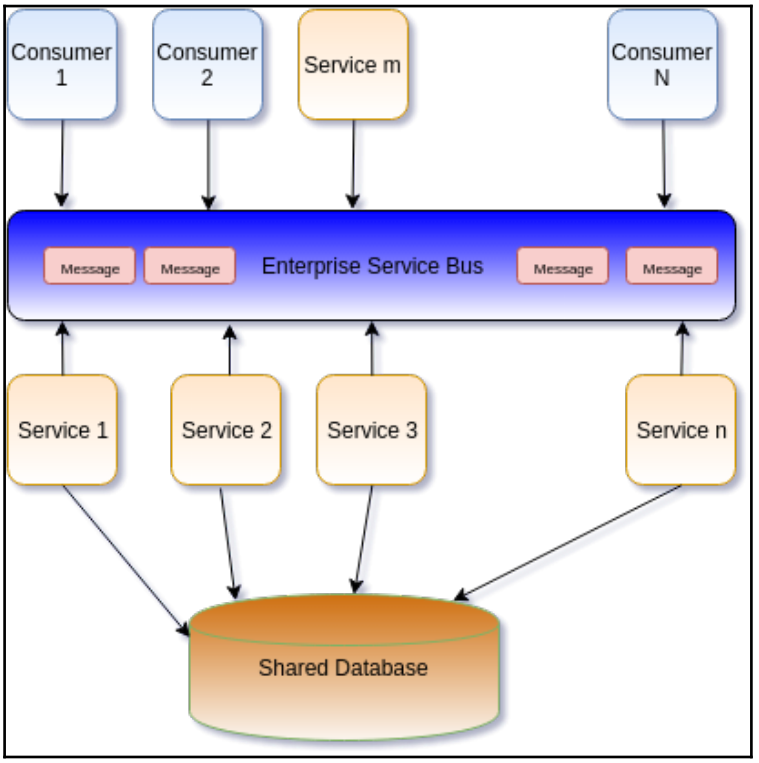
Although the situation is hypothetical, a similar kind of situation is encountered by most of the new joiners. Making their code live on the very first day increases the confidence of an employee. New joiners can be productive on the very first day, if they have to grasp less code base at the beginning. Understanding a big monolithic code base and having working knowledge of a fair number of modules may take a week's time or more by a new joiner before starting.

Microservice architecture brings many benefits to the organization. It breaks the whole application into smaller services, typically each in a different code repository. Smaller codebases are easy to understand. Microservices works on the isolation principle also, so deployment of each microservice doesn't affect the existence of other microservices. It makes applications language agnostic. Each service can be written in a different language with a different database (SQL, MongoDB, Cassandra, and so on). The application can accommodate a completely heterogeneous technology stack. This offers the developer the freedom to choose a language for their microservice.

Huh? Doesn't it seem like SOA?

The answer is one word, absolutely! Is there any difference between them? Again, absolutely! Microservice architecture is inspired by SOA. Their architecture seems like SOA, but there are differences that make microservice a more polished and organized version of SOA. In SOA also, problems are broken into services, and typically, they all used SOAP as a messaging language for communication. There are many consumers connected with many services through the service bus. Consumers can invoke any service by sending a message over the **Enterprise Service Bus** (**ESB**). Like in microservice, in SOA also, these services can be in the same language or a different language. SOA doesn't mention the bounded context of these services. Although these are not strict rules in microservices, microservice architecture gives some bounded context about these services in terms of SRP. The database in SOA is typically shared between the services, while in microservices, each service should have its own database and no other service can deal with that directly. SOA believes in the principle of sharing as much as possible, so SOA promotes sharing resources such as databases among different services, which actually increases the coupling between services. This contradicts with the microservice architecture. Microservice believes in the principle of sharing as little as possible. This changes the fundamental approach of designing any service in architecture.

Communication layer ESB, the middleware, in SOA becomes the single point of failure. In the case of failure in ESB, none of the services can work, consume any data, or talk to each other. Microservice deals with this problem with the failure in isolation principle. Typically, microservices talk over RESTful, and it eliminates the need for any heavy middleware communication channel. Another implication of SOA is service contracts. As there is no notion of bounded context in SOA, the same service can make changes in a different data model, which actually leads it to shared data model. This data model can have hierarchy or complex data types. Changes in any of this data model, even with a single field can result in changing the interface in all services and deploying the whole application. The following figure is showing the typical architecture of SOA Application:



Some of the problems mentioned above earlier with SOA can be handled by implementing the architecture in an efficient way, but not all the problems can be solved. SOA is a very broad term, and microservices can be a subset of SOA, which has some more fine- grained rule around the service- oriented design. There is a famous quote about Microservices, that is *"Microservices are, SOA done in better way".*

One major edge that microservice architecture has over monolithic applications is the speed to production environment. Microservices are getting popular not only in small startups but also in big companies. One of the biggest goals they achieve is shorter **Time to Market** (**TTM**). With dynamic and ever-changing requirements, TTM has become critical; product requirements change daily based on market reaction. Microservice architecture perfectly fits in this situation--speed, build, test, and ship. The market is very competitive these days and is evolving around customer feedback. The cycle is build, test, and ship, get the feedback from the market, build it again, test it again, and ship it again. If any product can be shipped to market in a shorter time with new features or upgraded with feedback from the market, then it will definitely have an edge over the competitior. Achieving the same in a monolithic environment is not easy; the whole application has to ship again even for a small feature.

Assorting large business domains into microservice components

Designing a robust microservices architecture starts by identifying the business domain, defining the capability around that domain, and then defining microservices around it. Breaking down a business domain can be achieved with the concept of **Domain-Driven Design** (**DDD**), an approach to software development for complex needs which connects the implementation to an evolving model.

The term DDD was introduced somewhere in 2004 with buzz keywords such as modeling, bounded context, entity, repository, and ubiquitous language. As DDD fits perfectly with the idea of microservices, this concept is again getting hype because of the popularity of microservice architecture. The philosophy of DDD is to try to understand the domain of a problem that we are trying to solve and model it around this domain. It suggests modeling on the basis of real business use cases. This modeling includes defining a set of concepts for domain, capability of domain, and bounded context around it. There are some basic concepts in DDD, which are as follows:

- **Ubiquitous language**: This means all the stakeholders (business, operation, developer) should share the common language (terminology) that should not be too technical, rather it should be more focused on the business side model. This language will be related to the bound context of domain.

- **Bounded context:** The main idea is to define the scope of a model, to define the clean boundaries of its context, and then do the most possible to keep the model unified. The defined model should be kept consistent and restricted within defined boundaries. Each bounded context should be independent to other context. It defines the separation of duties between the domains. Any particular business can have multiple domains in it such as shipping, inventory, sales, marketing, and so on. Each domain has its own capabilities. Each entity or component of a business remains in its own bounded context. DDD concept helps us identify these domains and their segregation point. This whole concept of DDD matches with microservice principles. Think around a model, encapsulate the model around bounded context, and these contexts define the boundaries.

As microservices should be isolated, it becomes more necessary for a microservice to have a clear responsibility or boundary. When adopting DDD, the bigger problem domain can be decomposed into smaller ones with bounded context, which will lead to a better understanding of defining microservices around them. If we don't do this clear boundary separation, we will have a leakage of responsibility in between microservices, and then we will have a big ball of mud without having any clear boundary. DDD helps in understanding and defining boundary context. A very good read on DDD is *Implementing Domain-Driven Design* by Vaughn Vernon ,which gives you a deep understanding of Domain-Driven Design.

Organizing microservice components around business capabilities

Microservices empower the organization with speed. They allow the organization to respond to the market in a quicker fashion. They improve the productivity, delivery, and, of course, the speed in delivering projects. The challenge in this format is to align your team inside the organization around this architecture efficiently. Microservices don't suit or fit every organization. Implementing microservice architecture can tweak the parameters inside the organization also.

Business capability can be defined as an independent operation service for a particular purpose in business. Business capability in the organization can be defined at each level/domain. The upper management level can have different capability definitions, operations can have different ones, sales can have different ones, technology can have different ones, and so on. So, identify the fine grain business capability and arrange the microservice around them.

Business capability and microservice seem to be the same by definition, but they is not. Running a promotional campaign is a business capability for the marketing department. After identifying this capability, start arranging the microservice component around it, such as email sender service, promotion link generator service, data collector service, and so on. It's a two-step process:

1. Identifying the correct business capability.
2. Defining the microservice component around it.

Both steps have to be done and fine grained as much as possible. This will help design the architecture efficiently.

Another important factor is communication. Communication is a live wire in any business and in any form. There can be communication between teams, between organization and end user to get constant feedback, communication between different levels of management (upper, middle, lower), and even communication between components of the software system of that organization. There is a law that relates communication between software components and how communication happens within the organization. The law is Conway's law.

As per Conway's law, *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure*.

This may seem hard to believe or may not be a valid statement to some people initially. Regardless of that, some of the well-known institutions such as MIT and Harvard take it seriously. They have done studies on this law, and compared some code based with the organization structure under which these codes develop. What they found is that this law works! Yes, indeed, 90 percent code bases in different segments, as they picked for their study, aligned perfectly to this statement. To understand it more let's see how it is proven in a monolithic application case. If you have one big developer team that is co-located in the same building, you will end up in a monolithic application. On the other hand, if you have small teams, members of one team should be co-located, but teams may not necessarily be co-located. Then, you will end up having distributed or multi-module application architecture, as in the open source projects. Open source projects have a kind of distributed and modular architecture. They have a number of small teams (even with a single member in a team) rather than one big co-located team. Then, these teams are not co-located. This law doesn't tell us anything about microservices, but it tells us how the team structure or organization structure can affect application design.

As per the law, the application architecture will be a copy of your organizational structure, but there is a high chance that your organizational structure can also have an impact on your application architecture. Team locations and team communication impact the application architecture, which can result in an aligned organization structure.

If an organization has five teams with four team members in each team, we encounter a situation where it is not easy to break down a service to be handled by one team. The joining of two teams could be a solution here. This is basically changing the organizational structure around application architecture.

However, each company has its paradigm, business capabilities and objectives, dynamics, team structure, and communication methods. Implementing microservice architecture can change the organizational dynamics and structure. Any organization should study its own current structure and communication methods so that they can land in the best of their microservice architecture.

To go or not to go for microservices

Microservice architecture promises to bring all the exciting and fascinating features into a system, which was not easy with monolithic architecture. Microservice offers all benefits such as scalability, high availability, loose coupling between components, high cohesion, and so on but there are some challenges involved in getting into microservices architecture. The decision of migrating to microservices isn't easy. This is a, relatively new field and there is lack of experience in this field. Hopefully more and more people delving into this can drive some patterns in the architecture. There are many factors that an organization should consider before migrating to microservice architecture. Before taking a big step, any organization should consider following factors.

Organization buy-in

Make sure people who are going to work in this environment are educated about it. It is not limited only to technical skill set, but they also need knowledge of the concept on which they are working. Mostly, people or teams should adopt the full stack developer philosophy. Their mindset should be aligned in DevOps work knowledge. Developer works in microservice environment, have a complete ownership of their microservice. It is very necessary that there should be no boundary in their mind as developers such as front end developer, back end developer, DevOps member and so on. The team should be ready to do all stuff by itself. Although having DevOps is always a good thing, the team should be self-sufficient to handle these tasks.

Experience DevOps

Make sure organizations have a mature DevOps team and procedures in place to handle multiple releases happening in a day. When moving from monolithic to microservices architecture, each service needs to be monitored separately. It will be an additional responsibility for the infrastructure and DevOps team. If you have to ship fast and independently, then your DevOps and monitoring system should be attentive enough to go and rollback efficiently if anything goes wrong.

Analyze existing database model

Efficient decomposing of databases is a problem. Sometimes, a database is designed in such a way that it is hard to decompose it or it requires a major design change in DB tables. Is the organization ready to take that hit? Is it feasible to break down one big DB into much smaller databases that are logically and physically different from each other? As data is in the decentralized form and resides in different databases, it means there will be no physical connection at tables. It leads to major changes in code bases.

Automation and CI/CD

Automation testing tools and CI/CD tools such as Jenkins and Team City should be in place for frequent delivery or release of microservices. Each microservice has to have its own deployment pipeline and configuration management. This will again be an overhead for the DevOps team. TDD is very important in this aspect. Unit test cases and integration test cases should be in place to support automation testing and continuous delivery.

Integration

For having different pieces of an application running at different places (microservices), one needs a lot of integration work, to get the desired output. Integration work like communication among these services and lots of maintenance work will be overheads while moving towards microservices.

Security

With so many components talking to each other over APIs or over messaging, security becomes an important aspect to look out for. Credential-based or token-based microservices offer many solutions to handle the problems, but it will be an overhead for the developing and infrastructure teams to manage.

Any organization that is ready to give blood to microservice architecture should consider giving thoughts to the preceding factors. Successful implementation of this will give any organization the freedom to scale, high availability, and any time deployment and can achieve zero downtime.

Example of successful migration

Netflix is one of the big shots that has stepped into microservices from the traditional monolithic application in 2009. They were so amazed with their learning that they shared the experience at each step of this journey with the world. Today, Netflix has roughly 800+ microservices running, which serve around 2 billion requests daily, resulting in 20 billion internal API hits in these microservices.

When Netflix started moving from a monolithic to a microservices architecture, the term "microservices" wasn't around. Initially, Netflix used to have a big monolithic application; they started with moving the movie encoding code base, a non-customer-facing application, to small new microservices. This is the best approach while migrating to microservices architecture: *always move non-customer-facing components from monolithic to microservice architecture first*. The failure of these types of components has less impact on traffic and the customer journey on the production. After this successful exercise, they started moving customer-facing components to microservices. By the end of 2011, they were completely on AWS cloud with a number of microservices running. Netflix has open sourced some of the tools that helped them successfully migrate from monolithic to microservices architecture. These tools are accessible at `http://netflix.github.io/`.

Sample project (credit risk engine)

During the course of this book, we will evolve a credit risk engine. It will not be a basis of many complicated aspects taken into account for creating algorithms of risk analysis. Rather it will be a very simple approach on 2-3 rules that will be simple to accept or reject for risk analysis. The focus of this application is not on how risk analysis is done, rather it is more focused on the factors such as how microservices should be designed, secured, communicate with each other, and so on.

While developing this kind of project, our development environment will include Spring Boot, and shipping microservices will be with Docker.

Spring

Spring is an amazing framework. Those who are working in Java and have worked with Spring in the past or are working with it presently already know its benefits. Those who are in Java but didn't work in Spring yet should consider it seriously. Spring is a lightweight framework that basically works on plain POJO objects with the dependencies injection principle. The responsibility of creating a new bean or object is taken by the framework itself. The life cycle of these types of objects is maintained by the framework. It encourages the modular approach in programming. AOP and transaction management are some of the essential bits in any project; these are provided by Spring with the boiler plate code available.

Architecture wise, Spring has a core container and many other projects to support various needs. Spring core container includes the following modules:

- Spring code module
- Spring bean modules
- Spring context
- **Spring Expression Language** (**SpEl**)

On top of these core modules, Spring has many other modules for making a robust Java-based application. Modules for database access, transaction management, security, Spring integration, batch processing, messaging, social plugin, Spring cloud, and so on; some of the modules that make code much simpler and easy to do efficiently. The benefits of Spring can take a whole book to discuss, and discussing the benefits of Spring is out of the scope of this book. So, in a nutshell, the Spring moulder approach is helpful for a Java-based developer to write a cleaner and efficient code. However, there is a small pitfall of Spring in terms of configuration. An average application written in Spring is likely to use more than four modules of Spring (core container, Spring MVC transaction management, database access). As the project grows, these module numbers can increase, and configuration and compatibilities between different versions of modules can become cumbersome. To make life easy for a developer, Spring Boot comes into the picture.

Spring Boot

Spring Boot comes as an advantage over Spring with bootstrapping code and ease of development, with avoiding the tedious configurations. Spring Boot is not another framework; it is just made on the top of the Spring framework to reduce the development effort. It saves a lot of effort for much of the boiler plate code, XML configuration, and annotation. It integrates with other Spring modules very easily. It also has an HTTP web server embedded in the web application. It has CLI. Spring Boot, which fits with many build tools such as Maven and Gradle perfectly.

Things are easy with Spring Boot!

A sample application developed in Spring Boot will provide better evidence to prove the preceding claim. Here, we will develop a very small application with one controller class that will greet the user with a name that is sent by the user in a URL.

The environment required for this sample project will contain the following:

- Java 1.8
- **STS** (**Spring Tools Suite**)
- Maven

Create a new Maven project in STS with the name `FirstBoot` and modify the `pom.xml` file of your project, as shown in the following code:

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

<modelVersion>4.0.0</modelVersion>
<groupId>com.sample</groupId>
<artifactId>firstboot</artifactId>
<version>0.0.1-SNAPSHOT</version>

<parent>
<groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-starter-
parent</artifactId><version>1.2.0.RELEASE</version>
</parent>
```

```
<dependencies>

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

 </dependency>
</dependencies>
<!-- more dependency if required , like json converter -->
</project>
```

The `spring-boot-starter-web` dependency brings the number of dependencies, including Tomcat webserver.

In the `/src/main/java/com/sample/firstboot/controller` folder, create a called `SampleController.java` file. This will be the controller of the application to handle the HTTP request coming to Tomcat. The contents of this file are as follows:

```
package com.sample.firstboot.controller;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
public class SampleController {

@RequestMapping("/greeting/{userName}")
String home(@PathVariable("userName")  String userName) {
return "Welcome, "+userName +"!";
}
}
```

A simple controller class would look like this. The `@RequestMapping` annotation identifies which URL this method is supposed to handle. It is supposed to handle the `/greeting/{userName}URL, then}`. Then, it will return a string with the `Welcome` and `userName` values. By default, it is for HTTP `GET` method. This is an example of a very small and straightforward controller class.

In the next step, create a new file with the name `SampleApplication.java`, with the following content:

```
packagecom.sample.firstboot.controller;

importorg.springframework.boot.SpringApplication;
importorg.springframework.boot.autoconfigure.SpringBootApplication;
```

```
importorg.springframework.context.ApplicationContext;

@SpringBootApplication
public class SampleApplication {

public static void main(String[] args) {
ApplicationContextctx = SpringApplication.run(SampleApplication.class,
args);
System.out.println("Application Ready to Start. Hit the browser.");

}

}
```

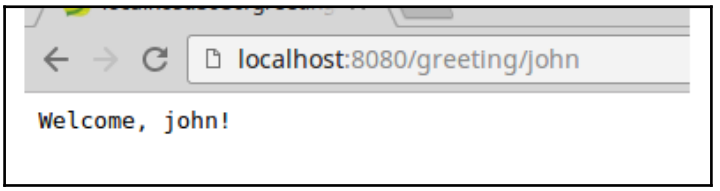
The `@SpringBootApplication` annotation replaces the different annotation required in the Spring framework. It removes the use of `@Configuration`, `@EnableAutoConfiguration`, `@Component` annotations.

Now, to run the code, open the console and traverse to the path of the application and type the `mvnspring-boot:run` command.

This command searches for the file with the `@SpringBootApplication` annotation and runs the application. Successful execution of this command will result in the following line in the end:

```
Started SampleApplication in 2.174 seconds
Application Ready to Start. Hit the browser.
```

The application is ready to be tested. To test the sample application, open the browser and type `http://localhost:8080/greeting/john`. This a screen similar to the following one: the result will be as shown in the following screenshot:



The `localhost` address is pointing to the user's computer, and `8080` is the port where Tomcat starts by default in Spring Boot. The `/greeting/john` is the part of the URL we mention in our controller class. The name can be anything, such as `xyz`, in place of `John`.

So, writing an application with Spring Boot is very easy. It saves the developer worrying about configuration and dependencies. The developer can focus on developing business logic and can be more productive.

Summary

In this chapter, you learned about microservices and their architecture. You also understood the benefits we can enjoy while using microservice architecture and also the tradeoffs of those benefits. In the upcoming chapter, we will define a sample project risk-assessing system that will have microservice architecture. This system will evolve with each chapter of the book. The risk-calculating project will use Spring Boot and Docker for development and deployment. Both these technologies were introduced in this chapter. What's next? The next chapter will explain defining microservices and the discovery of the service. Also, we will define our first microservice of user registration with Spring Boot.

2
Defining Microservice Components

In the last chapter, you learned about microservice architecture and understood different aspects of using it. As mentioned earlier, it is not a silver bullet for the entire problem, and even it cannot be implemented directly to every organization. An organization has to study many things before jumping to the decision to use this architecture. After all the studying, if any organization concludes to go into the microservice architecture, then they will face the next problem: how to define a microservice?

Definition of a microservice

There is no single universally accepted definition of microservices. Wikipedia defines microservices as follows:

> *Microservices are a more concrete and modern interpretation of service-oriented architectures (SOA) used to build distributed software systems. Like in SOA, services in a microservice architecture (MSA) are processes that communicate with each other over the network in order to fulfill a goal.*

There is no clear description on how to define microservice. Many people have expressed their different views in this area. Most of them conclude them into characteristics. The main characteristics are single responsibility, loose coupling, high cohesion, serving business purpose, lightweight communication, and few LOC (10-100). Although the LOC rule cannot be justified, as every language has a different syntax or library; so LOC can vary in that.

Amazon made a term famous: *two-pizza team*. As per this term, a microservice can be so big that it can be handled by a team which can be fed two pizzas. So, at max, 12 people can be in the team. It includes tester and developer, and if you are in agile, it should include a product owner. While defining a microservice, keep in mind that the attributes which change together should stay together. Another definition of microservice can be defined as a software application whose independently deployable, small, modular services communicate through a well-defined, lightweight mechanism to serve a business goal.

How small or big should it be, is the question for the industry. More and more people coming into the industry with different flavor of microservice implementations may give some direction to microservice design patterns. For now, I would like to spot it as an independent deployable module with an isolation failure plan serving a single business goal, which can be stated in simple short lines without using any conjunctions.

In the upcoming section, we will discuss two important aspects: **Service Discovery** and **API Gateway**. These two aspects play an important role while defining a microservice.

Service Discovery and its role

Microservices have to communicate with each other, and to do that, services need to find each other's location. One can bundle the IPs and URL path inside the microservice. This will cause a problem if the IP of another microservice changes. Any change in the IP or any service requires it to make a change in the code of the microservice using it and forces the release of that microservice. A solution for that is if the IP address is extracted from code and placed in an external configuration; we will talk more about the externalization of microservices in the next section. If you have a few services in your bag, it can solve our purpose. As your system starts evolving, then the number of microservices will increase. With an increasing number of microservices, the environment becomes more and more dynamic. Microservices can be coming up and down, assigning IPs on the fly based on traffic's need. This will make the process of locating a particular service or machine more and more difficult. Also, monitoring these increasing microservices will lead you to another challenge. In this situation, identifying the location of any particular service will not be an easy task and book keeping will not work here. This raises the need for a mechanism to locate any particular microservice in our system. That mechanism should give developers confidence to start and stop services in any order, and reconfigure services easily. To achieve this, the mechanism or external service should be capable enough to provide information about microservices coming up and down dynamically in the system. There are a few ways to solve this service location problem; one has to study enough to find the best suited method of their microservice architecture.

DNS

The first solution that comes to mind is DNS. DNS maps the name with the IP address. Multiple addresses can be mapped to a single service. But, what happens when a new service spins up? We will have to add an entry in the DNS. Additionally, if we are changing the IP address of any service, the change entered in the DNS will not reflect immediately. It has to wait for TTL to reflect everywhere. DNS has the advantage of being simple and easy, and it can be used for this, but it doesn't give the flexibility for the dynamic microservice environment. Spinning up multiple copies of services is a common scenario in companies where incoming traffic is big. In this kind of environment, if any service wants to talk with a particular instance of any other service which has dynamic scaling in place, DNS will not be very helpful. As instances are coming up and down at will, maintaining their entries through a DNS will not be very instant.

Discovery service need

After looking at the downfall of the DNS performance in a dynamic environment, there is a need for a service that is dedicated to this particular purpose, and it should provide the following facilities:

- Services should be able to register or deregister their availability
- Services should be able to communicate their changes to other services
- Services should be able to locate a single instance of a particular service

These services are provided by a discovery service. A discovery service is kind of a database of service instances available in a system. Each microservice that comes up and goes down should register or deregister here. Any microservice can get the meta-information, which can include host, port, service offer, and other details about any other microservice instance. This discovery service will be dynamically updated, as every service is registering itself here and this discovery service provides the particular instance of a service for which the calling service has raised the request. The requested service can be running in the same machine or different, same VPC or different, same region or different, and the calling service need not be aware of that. In addition to availability, microservices are also responsible for notifying the discovery service of any change in their metadata such as a change in service name or service status, whether it is master or slave, and so on. There are some solutions available in the market than run as a discovery service in the customer's platform and keep track of all the services in the environment. Some of the available solutions of discovery services are as follows:

- Etcd by CoreOS

- Eureka by Netflix
- Consul by HashiCorp
- Apache ZooKeeper
- SkyDNS
- SmartStack by Airbnb

They are not in any particular order, but readers are encouraged to explore Consul, ZooKeeper, and Eureka. As all these three are well supported in Spring, so developers familiar with Spring can easily get started on these in no time.

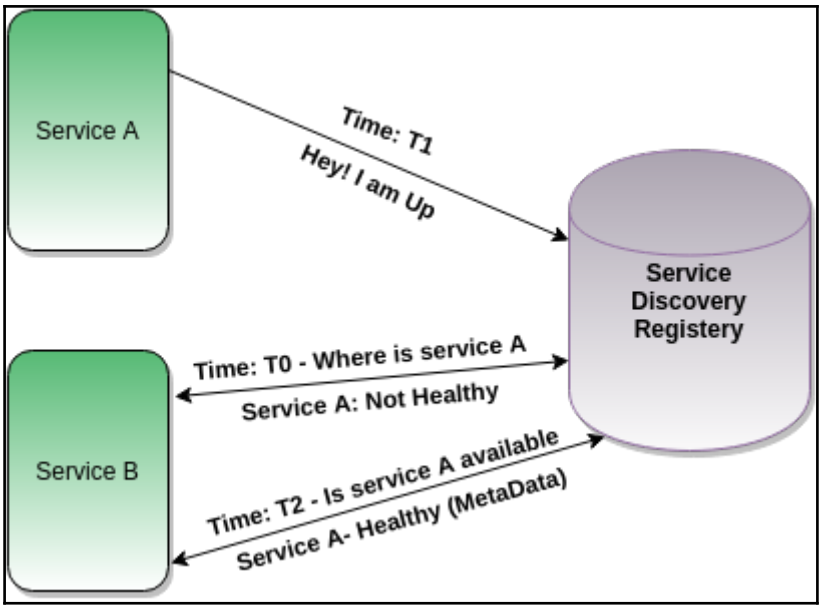
Registry/deregistry of services

To make this discovery service pattern successful, the main ingredient is to keep data in this discovery service updated. Any new service coming into existence or one that is destroyed or pulled out for maintenance should be aware of every event happening with its client. The following are the two types of registering mechanism:

- Self-registration
- Third-party registration

In self-registration, each microservice coming up in the system should register itself to the discovery service. While shutting down, it should also notify the discovery service. With these, the notification discovery service marks services as available or not available. Self-registry looks like a simple and clean solution for this, but it contradicts the single responsibility principle. Microservice will have one more responsibility of notifying the discovery service about its events.

The following diagram explain a method of self registry. In this, **Service B** is looking for the **Service A** instance and sending a request to the discovery service. When **Service A** comes up, it register itself. After that, service discovery is sharing the status and meta data of **Service A** with **Service B** when available:



In third-party registration, a separate process along with service discovery is responsible for keeping an eye on all the microservices running. This new process uses different mechanism to check the healthiness of the microservice such as poll, heartbeat, or hitting microservice health URL, and updates information in it. It should have some mechanism in place to detect the failing of any microservice.

Discovering other services

Other than registering a service discovery, the service has one more goal: providing meta-data about other services in the system. At any point, if any service wants to communicate with another service, it should be able to locate that service from this discovery service. The following are the two patterns of communication between services asking for any other service location and the discovery service:

- Client-side discovery
- Server-side discovery

In the client-side discovery pattern, each microservice will communicate with the discovery service directly. In this pattern, the service gets the network location of all other services and their instance location from the discovery service before making the request to the required service. After getting the locations, the calling service is supposed to have smartness in choosing the proper instance of the required service by implementing some load balancing algorithms. Netflix Eureka is a good example of using this pattern. We have a code example of Eureka in Spring in the next section. Ribbon is another tool that works with Eureka and helps the service to efficiently choose an instance for load balancing the request.

The client-side discovery pattern is simple and easy to implement with so many available tools. As in self-registration, there is a violation of the microservice's basic single responsibility rule. Now, each service is also responsible for finding the location of other microservices in the network, and also with some algorithm, it should make a request to a particular service instance to balance the traffic load. Still, organization uses this pattern for ease of use.

In the server-side discovery pattern, services will make a request to a single middleware entity, such as a load balancer or gateway. This gateway will communicate with the discovery service to locate the required service, and use logic to find the exact instance to talk with. The API gateway is supposed to have the logic of managing the traffic load and choosing the best instance of that particular service. It is aligned with a single responsibility rule for microservices. Microservices can focus on their business logic itself. We will discuss the API gateway pattern in the next section.

In spite of what pattern you choose, these services can be more useful with the information they provide. Along with healthy or not healthy, these discoveries can provide information about which instance is master and which instance is salve. Depending on the type of operation (read or write) a service wants to do on another service, it can choose a better one. Discovery service can also give information about which version it is running. If any X service is running two different versions of itself and a Y service wants to call service X, then discovery service will provide information about both the running versions and gateway, or service Y itself can choose the best version of X to talk, or service Y can also send information with a required version of X. Discovery service will return the required version. This feature can help in running different versions of a service simultaneously in the same environment.

Discovery service can help in reducing the network latency in call. As shown in the previous example, while requesting the location of service X, discovery service can provide that particular location of instance which is situated very near to service Y. Required version of service X can be located in the same rack, VPC, region, or zone. So, discovery service can have this intelligence to locate the best service instance to communicate, which eventually helps in the network latency.

Discovery service took the responsibility of finding healthy instance of required service, could be physically located near to asking service.It also took load balancing in consideration, it will pick least loaded service first. So, it gives more flexibility in a dynamic environment with minimizing the fear of failure. In the next section, we will be looking at a very small example on how discovery services works.

Example of service discovery pattern

There are many implementations available for this pattern, some of which are mentioned in the previous section, but for this particular example, we will use Netflix's Eureka. Spring has an in-built support for Netflix Eureka with Spring Boot (1.4.0+). With use of some annotation, we can create registry service and clients very easily. In this example, we will create one registry service and one client who will register itself while coming up. We will generate a sample code for Eureka server first from the Spring initializer or the Maven archetype. For this particular service, we will need some dependency to define `spring-cloud-starter-eureka-server`. Pom dependencies of eureka will be look like the following:

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

Spring also has an implementation for zookeeper and consul. This can also be easily embedded as a `spring-cloud-starter-consul-discovery` dependency or a `spring-cloud-starter-zookeeper-discovery` dependency.

In your main application class, the user has to add an `@EnableEurekaServer` annotation. There's nothing fancy going on in the following piece of code:

```
package com.practicalMicroservcies;

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

@EnableEurekaServer
@SpringBootApplication
public class EurekaServerDemoApplication {

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

The `@EnableEurekaServer` annotation enables the Eureka service discovery server. In `application.properties`, which will be located at `src/main/resources/application.property`, mention the following property:

```
server.port=8761
```

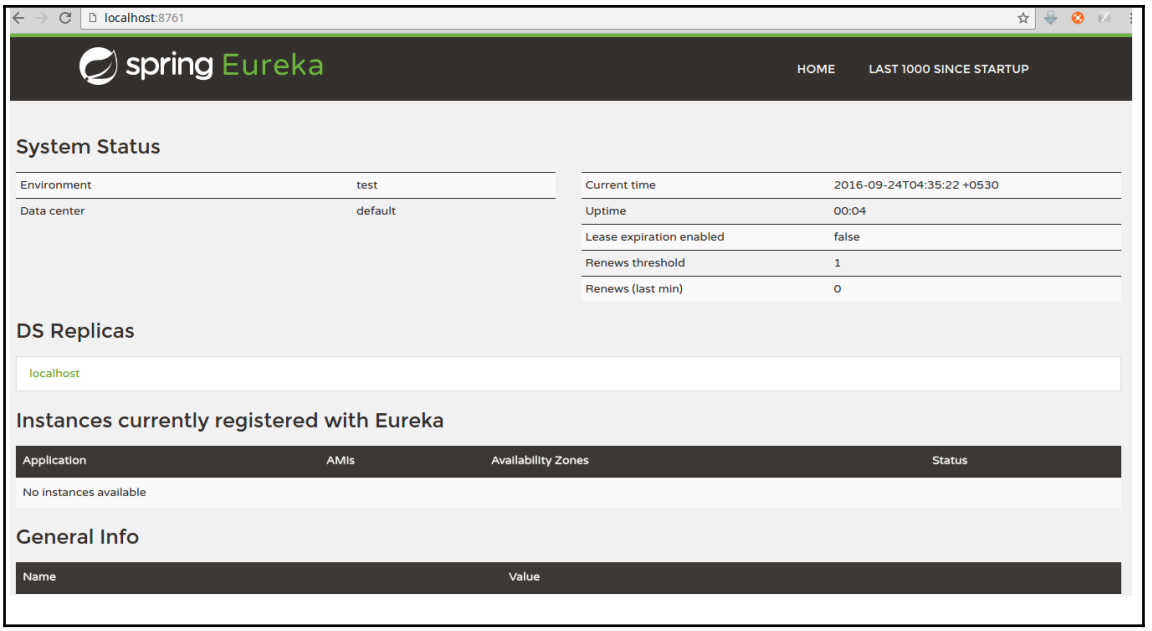
Run the application with `mvn spring-boot:run`; users are supposed to get a warning level like this:

```
WARN 24432 --- [nfoReplicator-0] c.n.discovery.InstanceInfoReplicator     :
There was a problem with the instance info replicator
```

This happens because, by default, the replica mode is on in the Eureka server, looking for another replica node, and it can't find any. Also, one sees the attempt of service to self-register. So, by adding the following two properties, these problems should be resolved:

```
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

After adding these properties, the server should be up and running on port `8761`. A dashboard can also be seen in the browser if you type `http://locahost:8761` in the address bar you will see the following screenshot:



As you can see, there is no entry under the heading instances currently registered with Eureka; this is because we don't have any service instances registered with this server. Other than this, some general information regarding servers is available under the **General Info** heading, such as CPU, memory, IP address, and so on. So, by adding single annotation and some properties, we have our service registry server up.

To create a client again, generate the template code for Spring Boot with application/project name `EurekaClientApplication` as follows:

```
package com.practicalMicroservcies;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@EnableDiscoveryClient
@SpringBootApplication
```

```
public class EurekaClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaClientApplication.class, args);
    }
    @RestController
    class ServiceInstanceRestController {

        @Autowired
        private DiscoveryClient discoveryClient;

        @RequestMapping("/Demo/{UserName}")
        public  String serviceInstancesByApplicationName(
                @PathVariable String UserName) {
            return "Hello "+UserName;
        }
    }
}
```

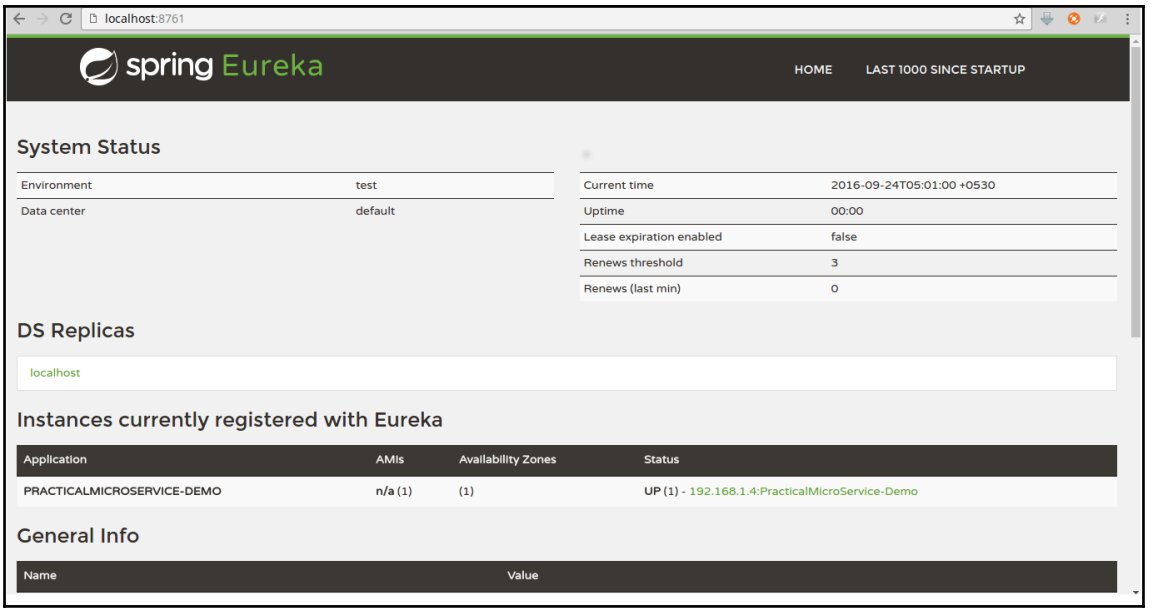
First to test the URL you can hit `http://localhost:8080/Demo/Alice` it will show `Hello Alice` as response.

Using the `@EnableDiscoveryClient` annotation, as soon as the client comes up, it looks for service registry and registers itself with the name mentioned in the `application.properties` file with the `spring.application.name` key. In our example, we have used the `PracticalMicroservice-Demo` name. In client logs, you should be able to see the following log line:

```
c.n.e.registry.AbstractInstanceRegistry  : Registered instance
PRACTICALMICROSERVICE-DEMO/192.168.1.4:PracticalMicroService-Demo with
status UP (replication=false)
```

We can see the same name as the registered service in the server console, as shown in the next screenshot. You can get any service instance with the following code:

```
this.discoveryClient.getInstances(serviceName);
```



By default, this implementation of the service client looks for the local host for service registry. Additionally, we can override this property by adding the service URL in the properties file. Registry of service with server can be slow because the server creates an instance of service and fills it with all the metadata that it gets from the service client. Even after registry, there will be a heart between server and client at the interval of 30 seconds; although, it is also configurable.

Spring has many excellent features to support microservice architecture and their pattern. With spring-cloud, it brings a numerous number of features to handle different cases that arise in microservice, such as Ribbon for a client-side load balancing (Eureka aware client-side load balancing), Hystrix (developed by NetFlix) for circuit breaker pattern, and so on. Discussing the implementation and the use of these technologies is not the scope of this book, so I would like to leave it to the reader to explore spring-cloud and its offering for microservice architecture.

Externalized configuration in the overall architecture

Location of configuration is another challenge for developers. Configuration burning inside the code is surely not a good idea. For each change in configuration, the developer has to make a new release; also, if some configurations are the same in multiple microservices such as the database server IP, these kinds of properties are duplicated in all the services. So, a better idea is to keep the configuration out of the code base. To implement this idea, we share a common location between microservices for properties, DevOps people can mount a drive on all the microservice instances. Using the NFS drive for this is an option. For each environment, there will be a different NFS server and drive mounted.

If you are on AWS, then mounting an S3 bucket in the entire microservice instance runs another option. Bucket can have some hierarchy to give customized properties to any microservice. Each microservice can have a different folder name and property files in it. The normal convention to save the property files is `s3://<bucketName>/<Environment_Name>/<microservice_name>`.

In this type of a scenario, there will be no track of change in the configuration. To solve this, we can introduce any version control system to maintain the configuration. Developers can commit the configuration in this repository and CI/CD tool while deploying a microservice deploy; they also deploy these configurations on the common sharing path. By looking at the log of version control, one can identify which configuration went with any specific version of microservice. Tagging the configuration with every release of microservices is a key. History is maintained now from version control, but usually organizations have multiple environments, and each environment has its own properties. This problem can be addressed by having multiple branches or multiple repositories per environment. In this case, the CI/CD tool should be aware of which environment it will deploy to, and from that particular repository or branch, it will deploy the configurations.

As this book is dealing with Spring Boot, let's discuss how we can achieve this in Spring Boot. Spring Boot has a particular order to search for the properties mentioned in the code. The order is as follows:

1. It looks for any command-line arguments.
2. On second step it try to find properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
3. Next step it try to find properties from JNDI attributes from default `java:comp/env`.

4. On fourth try it will look from `System.getProperties()`.
5. Next it try to look into operating system specific environment variables.
6. Then, From `RandomValuePropertySources`.
7. Profile-specific application properties packaged inside/outside your JAR (`application-{profile}.properties`).
8. Application properties inside/outside of your packaged JAR.
9. The `@PropertySource` annotations.
10. At Last spring boot try to find property values in properties specified using `SpringApplication.setDefaultProperties`.

We are not discussing all of the flow here, but readers are encouraged to read and try out the different ways mentioned here to try and find the best way to configure properties for their project. We are using `@PropertySources` in our sample project as well, so here is an example of how to use this annotation in code. Normally, this annotation is used in the configuration class, as follows:

```
@PropertySources({ @PropertySource("file:/home/user/microserviceA.props")})
```

In the preceding example, properties are loaded from the file whose path is mentioned as `/home/user/microserviceA.props`. Typically, this file can have a key-value-pair type of property defined that can be used in any of the beans in the application with a placeholder as follows:

```
@Value("${Connect.ip}")
private String connectionIp;
```

Using this statement, Spring looks into the earlier mentioned file and looks for the property like `Connect.ip=xx.xx.xx.xx`. This statement will take the value from property file into the `connectionIp` variable.

In this type of configuration, there is a problem of a fixed path mentioned in annotation. Using any environment variable, we can get rid of this static path. Let's suppose our configuration resides in `/mnt/share/prod/config` in the production environment, and `/mnt/share/stag/config` in the staging environment. Then our annotation will look like this:

```
@PropertySources({
@PropertySource("file:${CONFIG_PATH}/microserviceA.props") })
```

Here, `CONFIG_PATH` can be the environment variable and have values as per the environment. In Linux, it can be set as `export CONFIG_PATH=/mnt/share/stag/config`.

Sometimes, the developers have logically different configuration files in the same microservice; this annotation supports loading properties from multiple files as well. The following is an example:

```
@PropertySources({
@PropertySource("file:${CONFIG_PATH}/microserviceA.props"),
@PropertySource("file:${CONF IG_PATH}/connect.props") })
```

In the preceding example, properties will be loaded from both the files. For any placeholder, Spring looks for the property in both the files.

If we move towards the Spring Cloud with Spring Boot, it gives the flexibility of having the configuration server, which reads the property from any given URL. Any client service coming up will know where the configuration service is running. Client service asks configuration service if it has any properties related to this particular client. Then, the configuration server looks at the URI configured path for the file name that matched the client application name and sends back the JSON to the client. Configuration service is very easy to start; it just needs to use the `@EnableConfigServer` annotation. URI where configuration service looks for any asking property file is mentioned in the Spring Cloud property file as follows:

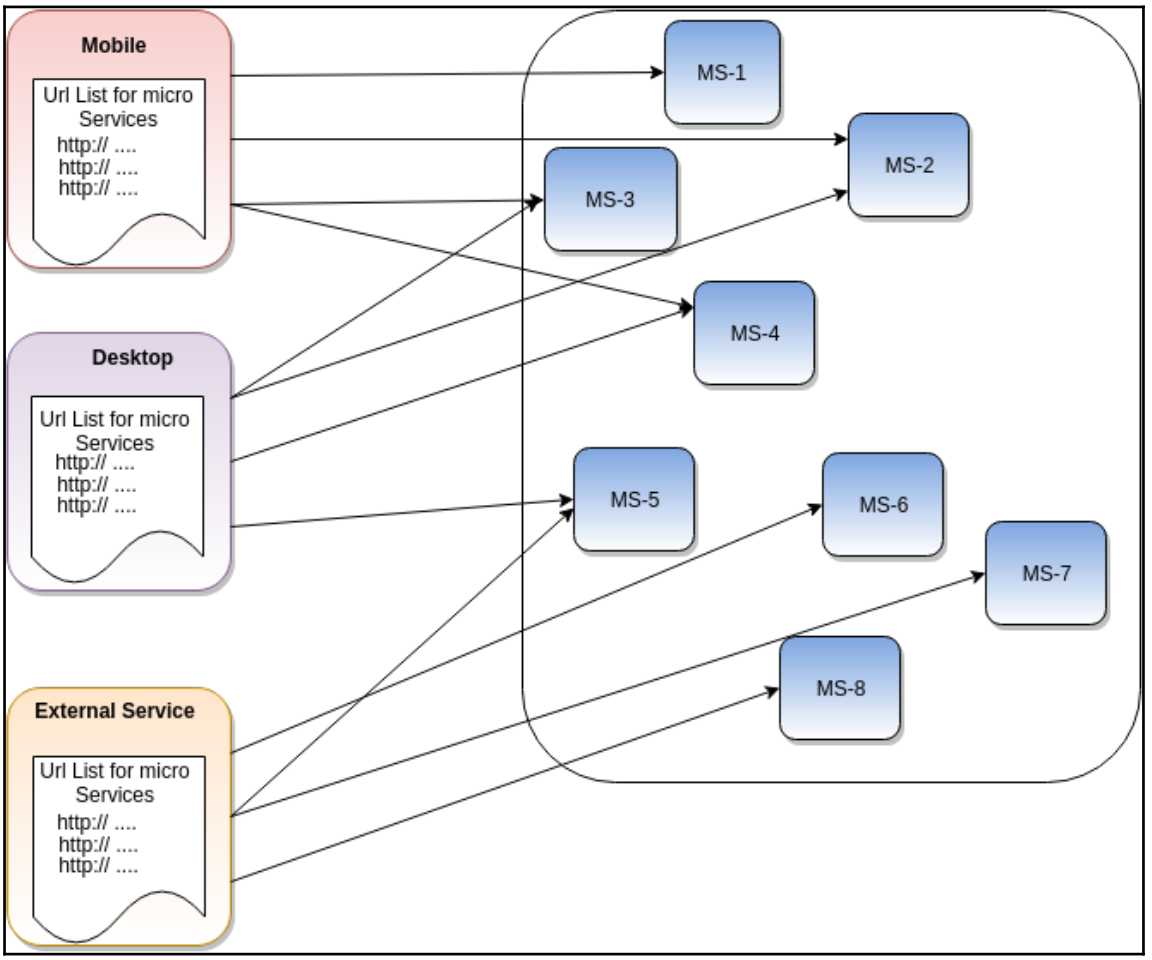
```
Server.port=8888
spring.cloud.config.server.git.uri =
  https://github.com/abc/microserviceA-configuration
```

The preceding file can also be in YAML format, the same way we defined the configuration service IP and port in the client configuration file. While client comes up and hits the configuration service to get configuration, configuration looks for the property name (`spring.application.name`) in the client configuration and for the `{clientName}.props` file in the mentioned location. In Spring Cloud, client can also be annotated as `@RefreshScope`. This annotation helps you dynamically load the property without any restart. It exposes the endpoint such as `<service endpoint>/refresh`, and hitting this endpoint will result in the recreation of bean.

In this section, we have seen many ways to achieve the externalization of configuration. However, I have only covered a couple of methods and left the others for you to explore. Having configuration service will help in dynamically changing the environment with hundreds of services running and sharing some common properties. Using this way, also increase the overhead work for the developers to maintain the configuration server and keep it running up. One can also use caching in front of the configuration server so that, in case of failure of the configuration service, caching can server request for some time. The API Gateway pattern can help in the externalization of some of the configuration of microservices. In the next section, we will discuss more about using the gateway pattern.

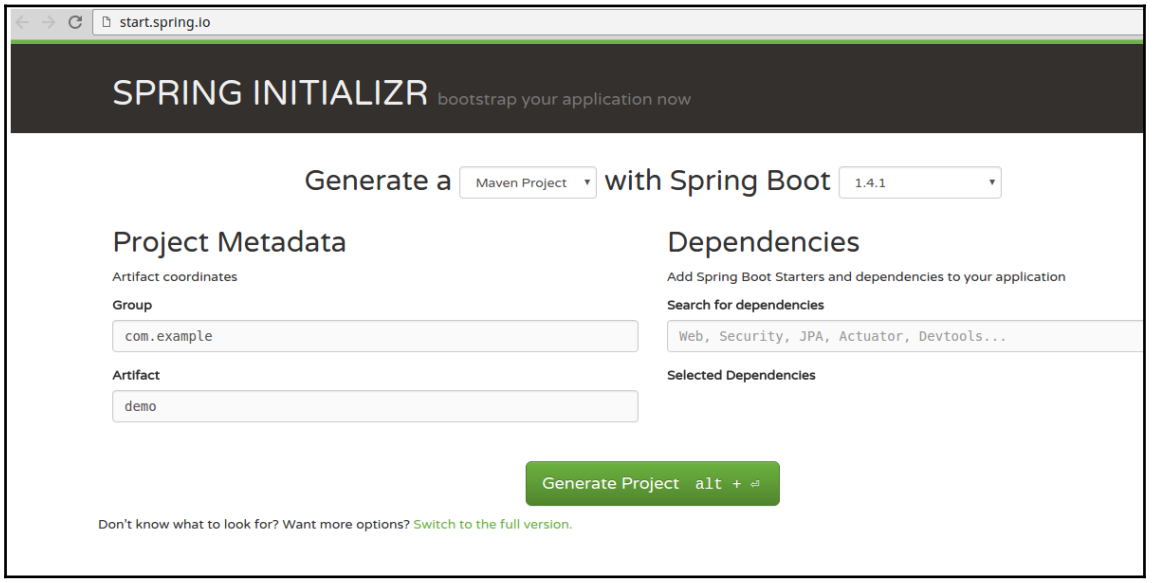
API Gateway and its need

Dynamic websites show a lot on a single page, and there is a lot of information that needs to be shown on the page. The common success order summary page shows the cart detail and customer address. For this, frontend has to fire a different query to the customer detail service and order detail service. This is a very simple example of having multiple services on a single page. As a single microservice has to deal with only one concern, in result of that to show much information on page, there are many API calls on the same page. So, a website or mobile page can be very chatty in terms of displaying data on the same page:



Another problem is that, sometimes, microservice talks on another protocol, then HTTP only, such as thrift call and so on. Outer consumers can't directly deal with microservice in that protocol. As a mobile screen is smaller than a web page, the result of the data required by the mobile or desktop API call is different. A developer would want to give less data to the mobile API or have different versions of the API calls for mobile and desktop. So, you could face a problem such as this: each client is calling different web services and keeping track of their web service and developers have to give backward compatibility because API URLs are embedded in clients like in mobile app.

Here comes the need of the API Gateway. All these problems can be addressed with the API Gateway in place. The API Gateway acts as a proxy between the API consumer and the API servers. To address the first problem in that scenario, there will only be one call, such as `/successOrderSummary`, to the API Gateway. The API Gateway, on behalf of the consumer, calls the order and user detail, then combines the result and serves to the client. So basically, it acts as a facade or API call, which may internally call many APIs. Following diagram show a way how API Gateway behaves:



The API Gateway solves many purposes, some of them are explained in the following section.

Authentication

API Gateways can take the overhead of authenticating an API call from outside. After that, all the internal calls remove security check. If the request comes from inside the VPC, it can remove the check of security, decrease the network latency a bit, and make the developer focus more on business logic than concerning about security.

Different protocol

Sometimes, microservice can internally use different protocols to talk to each other; it can be thrift call, TCP, UDP, RMI, SOAP, and so on. For clients, there can be only one rest-based HTTP call. Clients hit the API Gateway with the HTTP protocol and the API Gateway can make the internal call in required protocol and combine the results in the end from all web service. It can respond to the client in required protocol; in most of the cases, that protocol will be HTTP.

Load-balancing

The API Gateway can work as a load balancer to handle requests in the most efficient manner. It can keep a track of the request load it has sent to different nodes of a particular service. Gateway should be intelligent enough to load balances between different nodes of a particular service. With **NGINX Plus** coming into the picture, NGINX can be a good candidate for the API Gateway. It has many of the features to address the problem that is usually handled by the API Gateway.

Request dispatching (including service discovery)

One main feature of the gateway is to make less communication between client and microservcies. So, it initiates the parallel microservices if that is required by the client. From the client side, there will only be one hit.

Gateway hits all the required services and waits for the results from all services. After obtaining the response from all the services, it combines the result and sends it back to the client. Reactive microservice designs can help you achieve this.

Working with service discovery can give many extra features. It can mention which is the master node of service and which is the slave. Same goes for DB in case any write request can go to the master or read request can go to the slave. This is the basic rule, but users can apply so many rules on the basis of meta information provided by the API Gateway. Gateway can record the basic response time from each node of service instance. For higher priority API calls, it can be routed to the fastest responding node. Again, rules can be defined on the basis of the API Gateway you are using and how it will be implemented.

Response transformation

Being a first and single point of entry for all API calls, the API Gateway knows which type of client is calling a mobile, web client, or other external consumer; it can make the internal call to the client and give the data to different clients as per their needs and configuration.

Circuit breaker

To handle the partial failure, the API Gateway uses a technique called **circuit breaker** pattern. A service failure in one service can cause the cascading failure in the flow to all the service calls in stack. The API Gateway can keep an eye on the threshold for any microservice. If any service passes that threshold, it marks that API as open circuit and decides not to make the call for a configured time. Hystrix (by Netflix) served this purpose efficiently. The default value in this is a failing of 20 requests in 5 seconds. Developers can also mention the fall back for this open circuit. This fall back can be of dummy service. Once API starts giving results as expected, then gateway marks it as a closed service again.

Pros and cons of API Gateway

Using the API Gateway itself has its own pros and cons. In the previous section, we have described the advantages of using the API Gateway already. I will still try to list them in points as the pros of the API Gateway.

Pros:

- Microservices can focus on business logic
- Clients can get all the data in a single hit
- Authentication, logging, and monitoring can be handled by the API Gateway
- It gives flexibility to use completely independent protocols in which clients and microservice can talk
- It can give tailor-made results, as per the clients needs
- It can handle partial failure

In addition to the preceding mentioned pros, some of the trade-offs are also to use this pattern.

Cons:

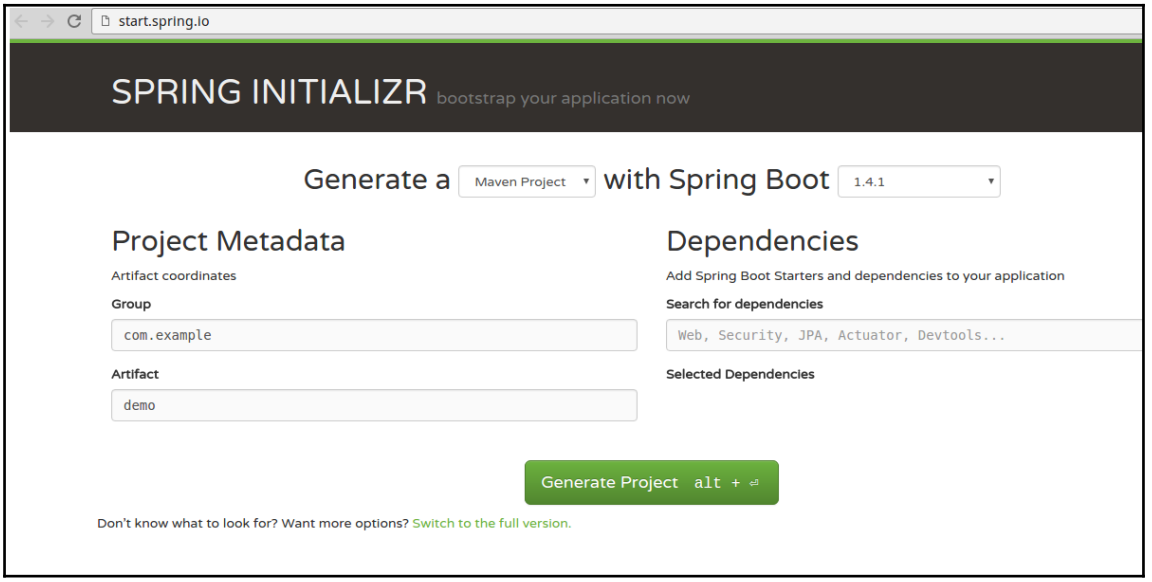
- It can cause performance degradation due to lots of happenings on the API Gateway
- With this, the discovery service should be implemented
- Sometimes, it becomes the single point of failure
- Managing routing is an overhead of the pattern
- Adding additional network hope in the call
- Overall, it increases the complexity of the system
- Too much logic implementation in this gateway will lead to another dependency problem

So, before using the API Gateway, both of the aspects should be considered. The decision to include the API Gateway in the system increases the cost as well. Before putting effort, cost, and management into this pattern, it is recommended to analyze how much you can gain from it.

Example of API Gateway

In this example, we will try to show only sample product pages that will fetch the data from the service product detail to give information about the product. This example can be increased in many aspects. Our focus of this example is to only show how the API Gateway pattern works; so we will try to keep this example simple and small.

This example will be using **Zuul** from Netflix as an API Gateway. Spring also had an implementation of Zuul in it, so we are creating this example with Spring Boot. For a sample API Gateway implementation, we will be using `http://start.spring.io/` to generate an initial template of our code. Spring initializer is the project from Spring to help beginners generate basic Spring Boot code. A user has to set a minimum configuration and can hit the **Generate Project** button. If any user wants to set more specific details regarding the project, then they can see all the configuration settings by clicking on the **Switch to the full version** button, as shown in the following screenshot:



For Zuul, user have to click on **Advance** and check the **zuul** check box and hit generate project. Project should be starting to download by now.

If you open the `pom.xml` of the project downloaded you can see the `zuul` dependency in it as follows:

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

Now let's create a controller in the same package of the main application class and put the following code in the file:

```
@SpringBootApplication
@RestController
public class ProductDetailConrtoller
 {
    @Resource
    ProductDetailService pdService;

    @RequestMapping(value = "/product/{id}")
    public ProductDetail getAllProduct( @PathParam("id") String id)
    {
        return pdService.getProductDetailById(id);
       }
}
```

In the preceding code, there is an assumption that the `pdService` bean will interact with the Spring data repository for the product detail and get the result for the required product ID. Another assumption is that this service is running on port `10000`. Just to make sure everything is running, a hit on a URL such as `http://localhost:10000/product/1` should give some JSON as a response.

For the API Gateway, we will create another Spring Boot application with Zuul support. Zuul can be activated by just adding a simple `@EnableZuulProxy` annotation.

The following is a simple code to start the Zuul proxy:

```
@SpringBootApplication
@EnableZuulProxy
public class ApiGatewayExampleInSpring
{
    public static void main(String[] args)
    {
     SpringApplication.run(ApiGatewayExampleInSpring.class, args);
     }
}
```

The rest of the things are managed in configuration. In the `application.properties` file of the API Gateway, the content will be something as follows:

```
zuul.routes.product.path=/product/**
zuul.routes.produc.url=http://localhost:10000
ribbon.eureka.enabled=false
server.port=8080
```

With this configuration, we are defining rules such as this: for any request for a URL such as `/product/xxx`, pass this request to `http://localhost:10000`. For the outer world, it will be like `http://localhost:8080/product/1`, which will internally be transferred to the `10000` port. If we defined a `spring.application.name` variable as a product in the product detail microservice, then we don't need to define the URL path property here (`zuul.routes.product.path=/product/**`), as Zuul, by default, will make it a URL/product.

The example taken here for an API Gateway is not very intelligent, but this is a very capable API Gateway. Depending on the routes, filter, and caching defined in the Zuul's property, one can make a very powerful API Gateway.

Sample application used throughout the book

The sample application that we will be developing during this book is **credit risk engine**. A risk engine can be described as a set of rules whose outcome is the magnitude of risk, which it calculates after analyzing some of the given input parameters. Add the term *credit* to this, and we can define a credit risk engine as something that calculates the risk magnitude of any given credit transaction. Credit risk can be of many types, such as lending loans, mutual funds, bonds, equities, and so on. Lending money to an institute or a person always includes a factor of risk.

Basically, the credit risk engine takes some parameters and variables from the present and past about that person or institute, and on the basis of rules defined in the engine, gives the credit score. On the basis of this credit score, engines take a decision to give credit or not to give credit to that particular person. Banking and financial institutes lend money to borrowers. They all use some kind of risk engine to assess the risk behind any heavy investment in a person or in a market. They include as many parameters as they can in their engine to cover the analysis and every aspect of the risk involved in that particular investment.

In our current credit risk engine project, we will be focusing on lending a loan to a person who is trying to buy a product in the range of, say, 5K to 40K. At **ZestMoney** (`http://zestmoney.in`), I have seen many complex rules and algorithms to compute the magnitude of risk involved in lending, but for our project here, we will be focusing on three to four parameters only. On the basis of these parameters, our credit risk engine will make a decision to accept or reject the particular loan application.

During the course of this book, we will be developing the following services:

- User registration
- Financial detail
    - Income detail
    - Liability details
- Credit scoring

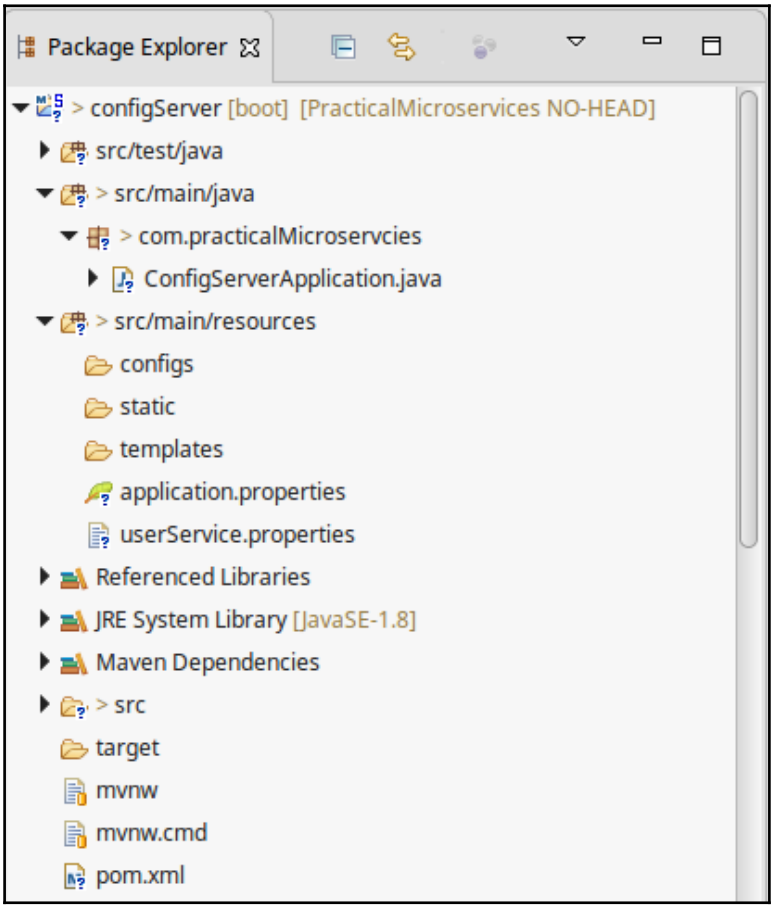
More details on these services will be in the relative sections/chapters.

Development of user registration microservice

In this section, we will launch the configuration service and main application.

Configuration server

To start with, go to `start.spring.io` and create a Spring Boot application, as mentioned in the earlier examples. For this project, the group name of our application will be `com.practicalMicroservcies`. First, we will create a configuration that will provide configuration to the various services of our project. To create this, we will create a Spring Boot project with `configServer` as the name and with an option of the cloud starter as well. If you download your project from `start.spring.io`, your folder structure will be as follows:



The `pom.xml` file will be as follows:

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

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.practicalMicroservcies</groupId>
    <artifactId>configServer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>configServer</name>
    <description>Configuration Server to provide configuration for
    different services</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-
        8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
        8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-config-server</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
```

```
        </dependency>

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

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>
.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

As you can see, we have added a `cloud-started` dependency in POM. Additionally, for cloud dependency, we will have a `Camden` client. At the time of writing this book, 1.4.1 release is the most stable version of Spring Boot, so we are using this version in the code.

The main application class code is as follows:

```
package com.practicalMicroservcies;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {
```

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

Nothing much here; the only change is the `@EnableConfigServer` annotation. This tells the Spring to treat this as a configuration server. Now, the next step is to define on which port this configuration server should run. Also, we need to define the various properties files, which will be queried by various microservices.

To define a port for this configuration server, we will have to mention value in the `src/main/resources/application.properties` file, as shown in the following piece of code:

```
server.port=8888
spring.profiles.active=native
spring.cloud.config.server.native.searchLocations=classpath:/
```

These properties tell the Spring about the port on which the server should run; in this case, it is `8888`. The next two properties defined are about the location of the properties files. As `native` is mentioned here, instead of the Git location, the config server should look for the property file in the search location mentioned in the `native` profile. The next property, `spring.cloud.config.server.native.searchLocations`, tells the Spring that all the property files should be on the class path. In this case, all property files will be in the `resources` folder, which, by default, will be on the class path.

The next step is to create the new properties files in resources. For now, we will be creating only one file with `userService.properties` as its name, as shown in the following block of code:

```
Connect.database=practicalMicroservice
spring.datasource.url=jdbc:mysql://localhost:3306/practicalMicroservice
spring.datasource.username=password
spring.datasource.password=password
# To generate SQL optimized for a Mysql
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL5Dialect
# Naming strategy, it can resolve the issue of uppercase table or column
name
spring.jpa.hibernate.naming-strategy =
org.hibernate.cfg.ImprovedNamingStrategy
```

How these properties will be in `userService` will be explained here. This completes our config server code. If you run the command from terminal, `mvn spring-boot:run`, you will see the application is up; and by writing curl command, you will get a property file result, something like this:

```
curl http://localhost:8888/userService/default
```

- `http://localhost:8888`: This is where the server is up and running.
- `userService`: This is the name of the service for which we want the properties.
- `default`: This is the active profile name. If we don't define any profile, it will always fall back to default.

The result of this command will be as follows:

```
{
  "name": "userService",
  "profiles": [
    "default"
  ],
  "label": null,
  "version": null,
  "state": null,
  "propertySources": [
    {
      "name": "classpath:/userService.properties",
      "source": {
        "spring.jpa.show-sql": "true",
        "spring.datasource.username": "root",
        "spring.jpa.hibernate.naming-strategy":
        "org.hibernate.cfg.ImprovedNamingStrategy",
        "spring.datasource.password": "root123",
        "Connect.database": "practicalMicroservice",
        "spring.jpa.properties.hibernate.dialect":
        "org.hibernate.dialect.MySQL5Dialect",
        "spring.datasource.url":
        "jdbc:mysql://localhost:3306/practicalMicroservice"
      }
    }
  ]
}
```

Under the `<source>` tag, you can see all the property we defined in `userService.properties`.

```
User service
```

User service will be responsible for all the operations related to a user. This includes registration, modifying, searching, and deletion. There will be a controller, repository, service, and model package, and there will be one controller to serve the request for user detail and address. The service exposed will be like adding, modifying, and deleting user details and adding user data. The records will not be actually deleted from the databases, rather they will just fill the `deletedOn` field in the database.

Table structure

The user service will use two tables. One to record the personal data of a user, such as their name, unique ID number, date of birth, gender, and so on and the second table to store the address of the user. To create a schema and table, we will use the following SQL file and `flywaydb`. **Flyway** is an open source database migration tool. It has Maven plugins that work well with the Spring Boot. To use the Flyway in Spring Boot, you will have to add the following dependency in the POM file:

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <version>4.0.3</version>
</dependency>
```

Flyway DB works on convention, so for any migration to happen, it will look for the file at the `src/main/resources/db/migration` path with a filename starting with `V1_0__DESCRIPTION.sql`. Flyway DB stores the schema revision in the database as well. It can read from the schema version which files were executed successfully and which are new ones to run. Lets create the `user_details` and `address` table now. The following is the SQL code to create the `user_details` table:

```
CREATE TABLE `user_details` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` char(36) NOT NULL,
  `first_name` varchar(250) NOT NULL,
  `middle_name` varchar(250) DEFAULT NULL,
  `last_name` varchar(250) DEFAULT NULL,
  `created_on` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  `deleted_on` TIMESTAMP DEFAULT NULL,
  `leagal_id` varchar(10) NOT NULL,
  `date_of_birth` TIMESTAMP NOT NULL,
```

```
  `gender` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `user_id` (`user_id`),
  KEY `user_details_userId_DeletedOn` (`user_id`,`deleted_on`),
  KEY `user_id_deleted` (`deleted_on`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

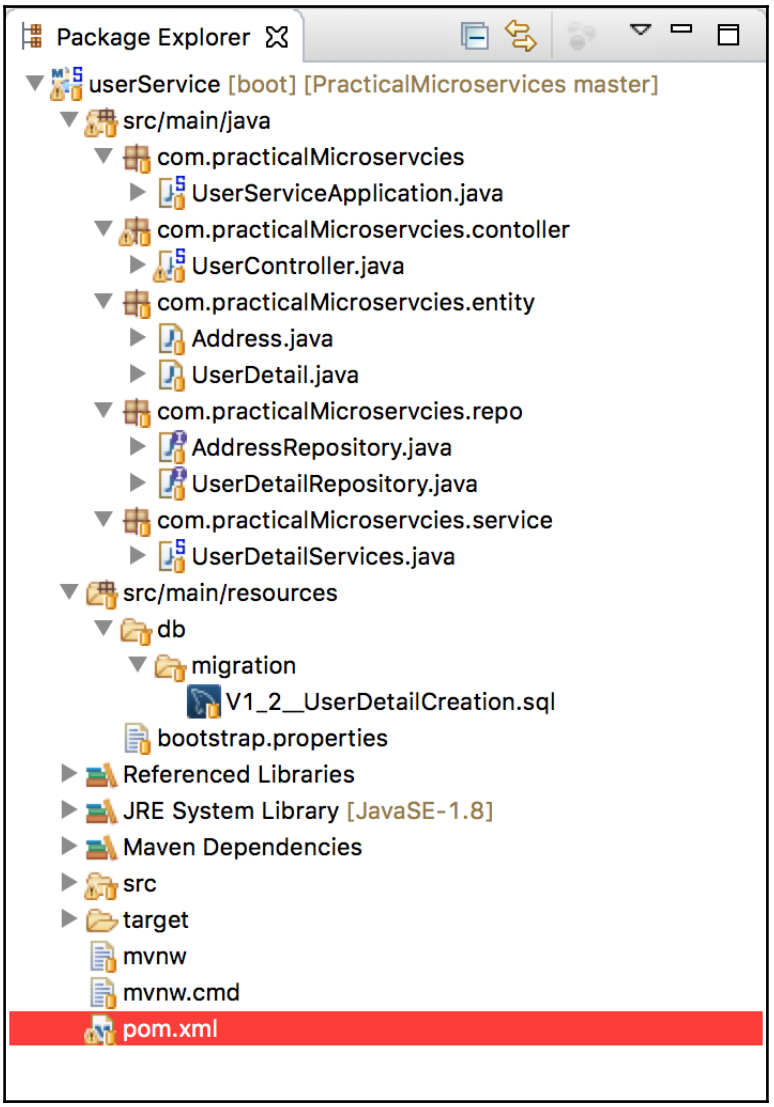
The SQL code for creating the `addresses` table, `table.user_id` is the ID from the `User_details` table:

```
CREATE TABLE `addresses` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` char(36) NOT NULL,
  `city` varchar(25) NOT NULL,
  `address_line_1` varchar(250) NOT NULL,
  `address_line_2` varchar(250) DEFAULT NULL,
  `pincode` char(6) NOT NULL,
  `created_on` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  `deleted_on` TIMESTAMP DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `user_id` (`user_id`),
  KEY `addresses_user_id_DeletedOn` (`user_id`,`deleted_on`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1;
```

As we did in the configuration service, a new boot application code with application name `userService` will be downloaded from `start.sping.io`. This time, we will also select the Spring data and MySQL connector to create a new project. For this we are using STS version 3.8.3.

After downloading and unzipping the code in a particular folder, you can run the `mvn eclipse:eclipse` command to generate Eclipse related files. After this, it will be easy to import a project into Eclipse, or STS as Maven project. After importing the project to your favorite IDE, you can see the file there.

Our final project structure will be similar to the following screenshot:



Lets work towards making our project in the shape of the preceding image. Lets start with POM file. Your POM will be as follows:

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

    <groupId>com.practicalMicroservcies</groupId>
    <artifactId>userService</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>userService</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-
        8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
        8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Camden.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-config-client</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.flywaydb</groupId>
            <artifactId>flyway-core</artifactId>
            <version>4.0.3</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Some of the new things that you will see here are Flyway dependency, `spring-data-jpa`, and MySQL connector in POM. Add the following code into your main application class:

```
package com.practicalMicroservcies;

import org.flywaydb.core.Flyway;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class UserServiceApplication {
```

```
    @Value("${spring.datasource.url}")
    private String url;
    @Value("${spring.datasource.username}")
    private String userName;
    @Value("${spring.datasource.password}")
    private String password;
    @Value("${Connect.database}")
    private String database;

    @Bean(initMethod = "migrate")
    public Flyway flyway() {
        /**
        As flyway Db need url seperetaly that is why we extracting only
        url from  given string
        */
        String urlWithoutDatabaseName=
        url.substring(0,url.lastIndexOf("/"));
        Flyway flyway = new Flyway();
        flyway.setDataSource(urlWithoutDatabaseName, userName,
        password);
        flyway.setSchemas(database);
        flyway.setBaselineOnMigrate(true);
        return flyway;
    }

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

    }
}
```

Here, we are using properties that are defined in the configuration server. We still have to define how this service will get those properties from the server. In order to do that, please add a `bootstrap.properties` file in the resources folder and add the following line to that:

```
spring.application.name=userService
spring.cloud.config.uri=http://localhost:8888
```

Both the properties are self-explanatory. The `spring.application.name` key will define the name of the current service. This name will communicate with the configuration server, and the configuration server will look for the property file with this name and serve it back. Bootstrap properties will be executed before anything is initialized in the project, so this placeholder for properties will always be filled soon.

Now, the property name that starts with `spring.datasource` is important. These three properties are used by Spring Boot to automatically create a data source without writing any code. The `connect.database` is not an internal property of Spring Boot. It can be any name to define the database name.

In `flywayDb` bean, we have set the schema name with this version of Flyway DB; it will create the schema if it is not already there.

So, if you run a command from the terminal now, `mvn spring-boot:run`, you should be able to see your database and table created in the MySQL.

Let's create the model classes for these two tables under the `entity` package. One will be `Address` and one will be `UserDetail`. Insert the following code in the `Address.java` file:

```
package com.practicalMicroservcies.entity;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "addresses")
public class Address implements Serializable {

    /**
     * Serial Id for serialization
     */
    private static final long serialVersionUID = 1245490181445280741L;
    /**
     * unique Id for each record.
     */

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    private int id;

    /**
```

```
     * User Id , unique for every user
     */

    @Column(name = "user_id", nullable = false, unique = true, length =
    36)
    private String userId;

    /**
     * City, Staying city of user
     */
    @Column(name = "city", nullable = false, length = 25)
    private String city;

    /**
     * Address Line 1 for User Address.
     */
    @Column(name = "address_line_1", nullable = false,  length = 250)
    private String addressLine1;

    /**
     *
     * Address Line 2 for User Address.
     */
    @Column(name = "address_line_2", nullable = false,  length = 250)
    private String addressLine2;

    /**
     * Pincode for user City
     */
    @Column(name = "pincode", nullable = false,  length = 36)
    private String pinCode;

    /**
     * Date on which user is created
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_on", columnDefinition = "TIMESTAMP DEFAULT
    CURRENT_TIMESTAMP", length = 0)
    private Date createdOn;

    /**
     * Date on which user is deleted.
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "deleted_on", columnDefinition = "TIMESTAMP DEFAULT
    NULL", length = 0)
    private Date deletedOn;
}
```

The setter and getter methods of the class variable should be created here. User can optionally over ride the `toString` method of class also, as follows:

```
@Override
public String toString() {
    return "Address [id=" + id + ", userId=" + userId + ", city=" +
    city + ", addressLine1=" + addressLine1 + ", addressLine2=" +
    addressLine2 + ", PinCode=" + pinCode + ", createdOn=" +
    createdOn + ", deletedOn=" + deletedOn + "]";
}
```

There will be one repository class for it, which is as follows:

```
package com.practicalMicroservcies.repo;

import org.springframework.data.jpa.repository.JpaRepository;

import com.practicalMicroservcies.entity.Address;

public interface AddressRepository extends JpaRepository<Address, Integer>
{
    Address findByUserId(String userId);
}
```

There will be an `Entity` class for `UserDetail` as well, which is as follows:

```
package com.practicalMicroservcies.entity;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "user_details")
public class UserDetail implements Serializable {

    /**
     *
     */
```

```
private static final long serialVersionUID = 4804278804874617196L;

/**
 * unique Id for each record.
 */

@Id
@GeneratedValue(strategy = IDENTITY)
@Column(name = "id")
private int id;

/**
 * User Id , unique for every user
 */

@Column(name = "user_id", nullable = false, unique = true, length =
36)
private String userId;

/**
 * First Name of User
 */
@Column(name = "first_name", nullable = false, unique = true,
length = 250)
private String firstName;

/**
 * Last Name of User
 */
@Column(name = "last_name", nullable = false, unique = true, length
= 250)
private String lastName;

/**
 * Middle Name of user
 */
@Column(name = "middle_name", nullable = false, unique = true,
length = 250)
private String middleName;

/**
 * Legal id Of user( could be social security number etc)
 */
@Column(name = "legal_id", nullable = false, unique = true, length
= 20)
private String legalId;

/**
```

```
        * Gender of user
        */
       @Column(name = "gender", nullable = false, unique = true, length =
       20)
       private String gender;

       /**
        * Date of birth of user
        */
       @Temporal(TemporalType.TIMESTAMP)
       @Column(name = "date_of_birth", columnDefinition = "TIMESTAMP",
       length = 0)
       private Date dateOfBirth;

       /**
        * Date on which user is created
        */
       @Temporal(TemporalType.TIMESTAMP)
       @Column(name = "created_on", columnDefinition = "TIMESTAMP DEFAULT
       CURRENT_TIMESTAMP", length = 0)
       private Date createdOn;

       /**
        * Date on which user is deleted.
        */
       @Temporal(TemporalType.TIMESTAMP)
       @Column(name = "deleted_on", columnDefinition = "TIMESTAMP DEFAULT
       NULL", length = 0)
       private Date deletedOn;
   }
```

We will be having the setter and getter methods of all the instance variables of `UserDetail`. We can have the `toString` method override also in the following class:

```
       @Override
       public String toString() {
           return "UserDetail [id=" + id + ", userId=" + userId + ",
           firstName=" + firstName + ", lastName=" + lastName + ",
           middleName=" + middleName + ", legalId=" + legalId + ",
           gender=" + gender + ", createdOn=" + createdOn + ", deletedOn="
           + deletedOn + "]";
       }

   }
```

The repository class for this entity is also required:

```
   package com.practicalMicroservcies.repo;
```

```
import org.springframework.data.jpa.repository.JpaRepository;

import com.practicalMicroservcies.entity.UserDetail;

public interface UserDetailRepository extends JpaRepository<UserDetail,
Integer> {
    UserDetail findByUserId(String userId);
}
```

Now, we will need a `service` class to deal with these repositories:

```
package com.practicalMicroservcies.service;

import java.util.Date;
import java.util.UUID;

import javax.annotation.Resource;
import javax.transaction.Transactional;

import org.springframework.stereotype.Service;

import com.practicalMicroservcies.entity.Address;
import com.practicalMicroservcies.entity.UserDetail;
import com.practicalMicroservcies.repo.AddressRepository;
import com.practicalMicroservcies.repo.UserDetailRepository;

@Service
@Transactional
public class UserDetailServices {

    @Resource
    AddressRepository addressRepo;

    @Resource
    UserDetailRepository userRepo;

    public void saveAddress(Address address) {
        addressRepo.save(address);
        System.out.println("User Saved!");

    }

    public void saveUser(UserDetail userDetail) {
        userRepo.save(userDetail);
        System.out.println("User Saved!");

    }
```

```
    public Address getAddress(UUID userId) {
        Address returnAddressObject =
        addressRepo.findByUserId(userId.toString());
        return returnAddressObject;

    }

    public UserDetail getUser(UUID userId) {
        UserDetail userObjectToRetrun =
        userRepo.findByUserId(userId.toString());
        System.out.println("User Saved!");
        return userObjectToRetrun;

    }

    public void deleteUser(UUID userId) {
        Address addressObject =
        addressRepo.findByUserId(userId.toString());
        addressObject.setDeletedOn(new Date());
        addressRepo.saveAndFlush(addressObject);
        UserDetail userObject =
        userRepo.findByUserId(userId.toString());
        userObject.setDeletedOn(new Date());
        userRepo.saveAndFlush(userObject);
        System.out.println("User Deleted!");

    }

}
```

Finally, we need a controller to handle the request from frontend. The user ID in this service will be a UUID and it is supposed to be generated at the frontend. By doing so, the frontend or consumer can hit the microservice in parallel using the same user ID, as shown in the following piece of code:

```
@RestController
@RequestMapping("/PM/user/")
public class UserController {


    @Resource
    UserDetailServices userService;

    @Resource
    ObjectMapper mapper;
```

We need to have two services here, one is `UserDetailServices` which we need to do user related operations and object mapper for JSON to object and vice versa conversion.

We need a method which should be responsible for adding a new address in the database corresponding to any user:

```
     @RequestMapping(method = RequestMethod.POST, value =
"{userId}/address", produces = "application/json", consumes =
"application/json")
    public ResponseEntity<String> createUserAddress(@RequestBody
    Address address, @PathVariable("userId") UUID userId) {
        logger.debug(createUserAddress + " Address for user Id " +
        userId + " is updated as " + address);
        address.setUserId(userId.toString());
        userService.saveAddress(address);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    / Below  Method is responsible for creating a user.

    public static final String createUser = "createUser(): ";

    @RequestMapping(method = RequestMethod.POST, value = "{userId}",
    produces = "application/json", consumes = "application/json")
    public ResponseEntity<String> createUser(@RequestBody UserDetail
    userDetail, @PathVariable("userId") UUID userId) {
        logger.debug(createUser + " creating user with Id " + userId +
        " and details : " + userDetail);
        userDetail.setUserId(userId.toString());
        userService.saveUser(userDetail);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }
```

The following is the delete request for deleting the user from the database based on ID:

```
@RequestMapping(method = RequestMethod.DELETE, value = "{userId}", produces
= "application/json", consumes = "application/json")
    public ResponseEntity<String> deleteUser(@RequestBody UserDetail
    userDetail, @PathVariable("userId") UUID userId) {
        logger.debug(deleteUser + " deleting user with Id " + userId);
        userService.deleteUser(userId);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }
```

The following method is responsible for getting the user detail from ID which is a `GET` request:

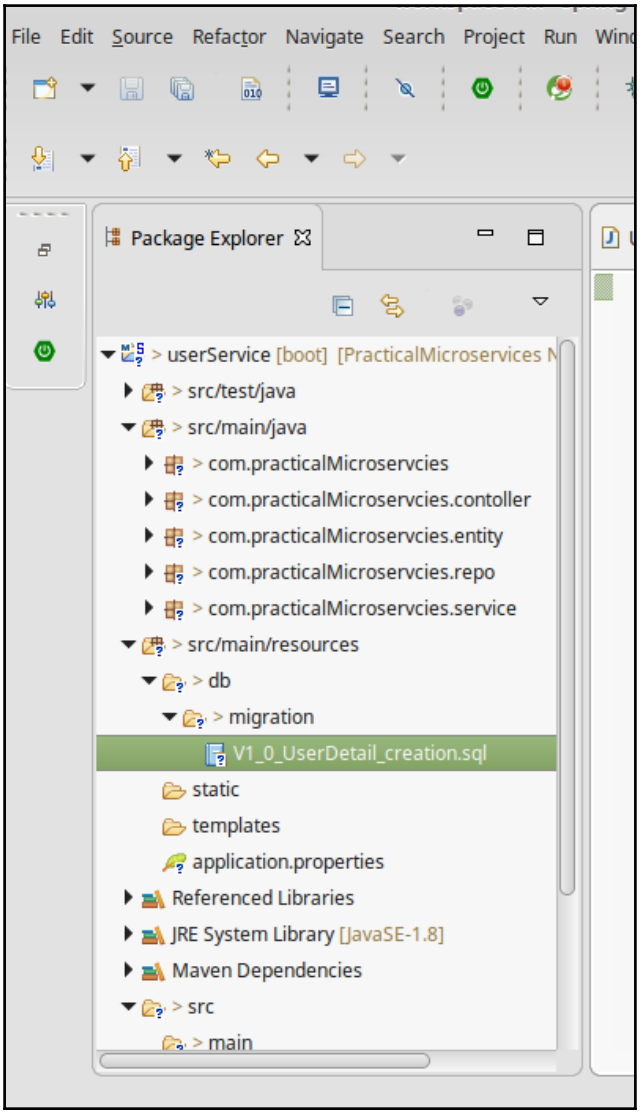
```
    @RequestMapping(method = RequestMethod.GET, value = "{userId}",
    produces = "application/json", consumes = "application/json")
    public ResponseEntity<UserDetail> getUser(@PathVariable("userId")
    UUID userId) {
        logger.debug(getUser + " getting information for userId " +
        userId);
        UserDetail objectToReturn = userService.getUser(userId);
        if (objectToReturn == null)
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        else
            return new ResponseEntity<>(objectToReturn, HttpStatus.OK);
    }
```

The next method is responsible for getting the address from the database respective of any particular user ID:

```
    @RequestMapping(method = RequestMethod.GET, value = "
    {userId}/address", produces = "application/json", consumes =
    "application/json")
    public ResponseEntity<Address> getAddress(@PathVariable("userId")
    UUID userId) {
        logger.debug(getAddress + " getting address for user Id: " +
        userId);
        Address objectToReturn = userService.getAddress(userId);
        if (objectToReturn == null)
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        else
            return new ResponseEntity<>(objectToReturn, HttpStatus.OK);
    }
```

The URLs exposed from this controller are as follows:

```
Post http://localhost:8080/PM/user/<user_Id>
Post http://localhost:8080/PM/user/<user_Id>/address
Get Post http://localhost:8080/PM/user/<user_Id>
Get Post http://localhost:8080/PM/user/<user_Id>/address
Delete Post http://localhost:8080/PM/user/<user_Id>
```



At this point, we are good to test a simple user service. Start the configuration server and user detail service in different terminals with the `mvn spring-boot:run` command. You will see that your database is created, and you can test to verify that the URLs are working fine with the help of any rest client, such as postman or curl. For address entity, try the `POST` request with a JSON value, as follows:

```
{
  "userId": "93a52ce4-9331-43b5-8b56-09bd62cb0444",
  "city": "New York",
  "addressLine1": "4719 Fermun Road",
  "addressLine2": "Yorktown Heights",
  "pinCode": "10004"
}
```

To create a user, use the following JSON:

```
{
"userId": "93a52ce4-9331-43b5-8b56-09bd62cb0444",
  "firstName": "John",
  "lastName": "Montgomery",
  "middleName": "Allen",
  "legalId": "B053-62-64087",
  "gender": "1",
  "dateOfBirth": "2010-10-27T11:58:22.973Z",
}
```

We start with a basic service will put security and testing bits in the respective chapter.

Summary

In this chapter, you learned about service discovery, externalization of configuration, and the API Gateway concept. We have also tried simple and short examples for each concept. Because Spring Boot is the code technology of this book, in our examples, we stuck to the implementation of these concepts in Spring Boot. Also, we have defined our first microservice, user registeration. It is in its initial stage, and it will evolve in terms of security, communication, and so on in the next chapter. In the next chapter, we will discuss how microservices should communicate with each other, and also our next service, financial service. We will also evolve the communication of a user registration microservice.

3
Communication between Microservices Endpoints

In the last chapter, you learned about service discovery and the API Gateway pattern. The chapter explained how to find microservices, but it would be interesting to dig into the detail of how microservices actually communicate at the protocol level. There are many options available for these interservice communication patterns. Once you decide which pattern suits your requirements, you will learn about the protocol that will be used. It could be SOAP, REST, XML-RPC, and so on. In this chapter, we will go through the major pattern found in interservice microservices communication. We will also go through an example of one of these patterns.

In this chapter, we will cover the following topics:

- How should microservices communicate with each other?
- Orchestration versus choreography
- Synchronous versus asynchronous communication
- Use of message brokers for event-driven microservices
- Development of a financial microservice

How should microservices communicate with each other?

Nowadays, a single page on an Android app or a web page shows lots of different categories of data. If any system is backed by microservice architecture, then it is not possible to generate different categories of data produced by a single microservice. Different microservices help in producing such categorized data, which leads to lots of communication between different microservices.

While dealing with a monolithic application, the whole code base resides at the same place and in the same language. So, communication between different modules happens through the function call or method call. However, this cannot be the case with microservices. Each microservice is a different process that runs in a completely different context or may be on different hardware. There are different perspectives to look at the inter-microservice pattern.

From one perspective, the flow of control and communication between microservices can be categorized into the following categories:

- Orchestration
- Choreography

Orchestration versus choreography

Think of a team that is working together towards one goal. There can be two situations here. The first one can be when the team has a leader and he is the brain of the team. He knows the capability of the team better, and he coordinates the work of different individuals in the team. When one person in the team finishes the work, the leader tells another person to start the work that was dependent on the first person's work. The second scenario could be when the team is self-motivating, talking to each other (may be via emails, calls, instant chat messages, or any other medium) and capable of working and communicating without any leader. In the first scenario, you have to hire a leader to guide the team, and it is a pain to find a suitable match. Then, you have to give a salary to that person. In the second scenario, you will save all this pain. However, the downside is if you have to change the flow, you have to inform every person and intimate to every new person after completing your task. In the first scenario, the leader can tell you what is happening in the team at any time, but in the second scenario, you have to interact with all team members to know the exact situation.

The first scenario is called orchestration, and the second scenario is called choreography. Now, let's discuss them in detail.

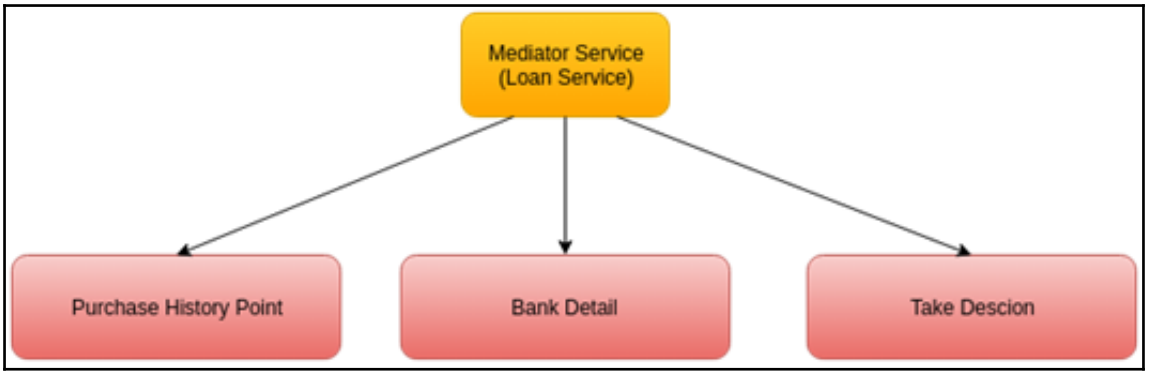
Orchestration

As the name suggests, this pattern is derived by one entity, like the conductor in an orchestra. The conductor's work is to ask the services to work as the signals, and the services will respond to that. Think of a team of musicians who are asked to perform. Everyone knows their jobs and will do it well, but how do they all know what beat or string to play at what time? With the hand flow of the conductor, each musician knows when to play which instrument, and it becomes a performance.

The same concept is applied here in the microservices pattern. There will be a mediator service that will act as the brain or conductor of the orchestra (number of microservices assembled). This mediator service will trigger and ask each microservice to start the action. All microservices should be working towards a single business purpose.

The mediator service works by sending messages to the desired services. These messages can be sent through HTTP calls or socket calls. The point to note here is that this pattern can be synchronous or asynchronous. Normally, in this scenario, individual services are not designed to talk to each other. Modifying the workflow can be easy with orchestration. In orchestration, the protocol should not be the boundary. The mediator service should be capable enough to talk to different services in a different protocol, such as with service A it is talking on HTTP and with service B it is talking with XML/RPC protocol. An intelligent API Gateway pattern explained in `Chapter 2`, *Defining Microservice Components*, can play the role of that mediator service.

There are challenges in this approach. Here, the mediator service becomes the central authority, and all the logic starts residing here, which is not a good microservice design. Logic inside the mediator service leads to lots of dependency between services. For example, there could be logic in the mediator service that if service A responds, then call service B. In this scenario, blindly calling service B is dependent on service A.



In the preceding example, the mediator service first checks its purchase history and bank details. Then, it asks the decision service to take a decision. One can always know the status of his or her application, but as mentioned earlier, logic is residing in the loan service now, which will start hitting the system while scaling.

Running, maintaining, and keeping this mediator service up is the cost of this pattern, but another thing is that it becomes the single point of failure in the whole architecture. This can be handled with better tooling and cloud-based techniques such as scaling, clustering, and so on. In addition to this, the orchestration model is not easy to implement in a complex domain. If you have one decision maker or controller, it becomes hard to follow the distributed design.

Choreography

Unlike with orchestration, in choreography, each component is tuned to carry out a predefined task in coordination with the others. It is more like a dance group that has been taught to perform predefined steps. Sometime, the dancers perform steps with other dancers or in parallel with other dancers.

Orchestration always represents control from one perspective. This differs from choreography, which is more collaborative and allows each microservice to be involved in the interaction.

In choreography, each service talks to another service after finishing its work or, sometimes, also between its work to trigger or initiate the task in the next service. This communication can happen in a one-to-one, one-to-many, or many-to-many fashion. Like in orchestration, choreography can happen in a synchronous way or asynchronous way. Ideally, in choreography, there is a global protocol on which each service decides to communicate with each other. Choreography is preferred to be used with asynchronous communication. As there is no central service, each component just raises the event when it finishes its task, which makes the services decoupled and independent of each other. Each component will be listening to an event or message raised by another service and reacting to those messages.

In choreography, the services are connected with messages or subscriber channels or topics to observe their environment. With message-based communication, choreography will be very useful. Decoupling, and adding a new service and removing old services is very easy. It saves the system from dependency hell, which can arise in the orchestration pattern. Needless to say, choreography increases the complexity in the system. Each service is producing and consuming messages at run time, so there is no proper way to identify the exact state of any transaction. A service has to have its own implementation to see the current state of the system. This could be a trade-off with this pattern.

No one pattern is better than another here. It is more of a situational call to use any of the patterns mentioned earlier. Sometimes, a mix of these two patterns works in the same system. The event stream for analyzing the big data can use a choreography pattern, and the system inside uses orchestration. Let's move to another angle to see the pattern in the microservice.

From another perspective, the method of data flow and communication between microservices can be categorized into:

- Synchronous
- Asynchronous

Synchronous versus asynchronous communication

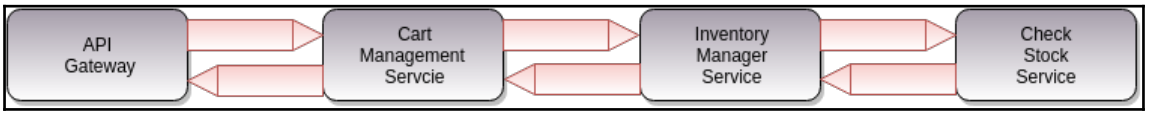
In the previous section, we saw how communication between microservices can be triggered. It can be one service controlling another service (orchestration), or a microservice raising its voice when it needs to talk to another microservice (choreography). One can choose any pattern there, but irrespective of what they have chosen, the other question is whether the trigger of communication should be synchronous or asynchronous. There are two other ways to look at how communication is done first: does your microservice want to collect data from other microservices by giving them conditions and waiting for a result? Secondly, they may want to broadcast a message, to tell every other microservice that it has done its part, to please take over from here, and start doing its own next task. In the upcoming section we will explore both of these methods.

Synchronous communication

As the name says, in synchronous communication, communication happens between two services, and a timely response is expected. The consumer server waits or blocks until the response from the remote server is received. In synchronous communication, receivers instantly know whether the call is successful or unsuccessful, and a decision can be taken on that. It is very easy to implement this approach. Because of the request/response architecture, REST becomes the best choice for synchronous communication. Although microservice developers consider JSON to be the best approach over HTTP, SOAP, XMLPRC, and socket programming are all candidates of synchronous communication. To understand this better, we will go through the example inventory management system.

Consider a sunglasses website `http://myeyewear.com/`, assuming that it is very popular for shades. Let's assume that the website has announced a Christmas sale. At a particular moment, there is the possibility of many people looking at one product and trying to add shades to their shopping cart. In this scenario, companies want to check whether the item is available and only then let customers add it to their shopping carts. Companies may also block the inventory for this product until either the customers buy it or their cart time expires. There are many ways to handle this situation, but for the sake of the example here, we will assume that the microservices are communicating over REST. This process will take place in the following steps:

1. The client/consumer goes to the API Gateway to add the product to their shopping cart.
2. The API Gateway goes to the cart management service with the product ID to add to the user's cart.
3. The cart service hits the inventory service to block the inventory of the given product ID.
4. At this level, the cart management service waits for a response from the inventory service. Meanwhile, the inventory service checks the availability of the product, and if found in stock, it blocks one and responds to the cart management service.
5. Based on the response received, the cart service adds the product to the cart and responds to the consumer.



In this communication, the calling service has to know the existence of the inventory service, which, as a result, creates some kind of dependency between them. Also, the calling service blocks the call until it receives a response from the called server. This is an example of tight coupling. The calling service has to handle various errors that are raised from the called service, or perhaps the calling service is down. Although timeouts are there to handle these kinds of situations, they are a waste of system resources. Also, the last few call systems get no result from the called service, but this time out will be there in each call and will limit the resource consumption and delay the response time.

A circuit breaker pattern helps deal with errors from the calling service or when there is no response from the calling service. In the previous chapter, we introduced the topic of a circuit breaker. This time, let's go into more detail.

A circuit breaker pattern identifies the failure based on some rules/threshold defined. Then, it prevents the calling service from calling the infected service again for a certain period of time, or it can go to any fall back method. In the circuit breaker pattern, there could be three states of circuit or call:

- Closed state
- Open state
- Half-open state

In the closed state, the call executes perfectly, and the calling service gets the result from the called service. In the open state, the called service does not respond and has crossed the threshold. In this case, that circuit will be marked open. It will not entertain more requests to the infected service for the configured amount of time and fall back to another configured method. There is the last state, which is the half-open state. In this state, whenever the configured time is over for the open state, the circuit breaker makes it the half-open state. In this state, the calls are passed to the infected service occasionally to check whether the issue is fixed. If it is a fixed circuit, it will be again marked as closed else it will revert back to open state.

To implement this pattern, interceptors are used at the side of the calling service. The interceptor keeps an eye on every request going out and responses coming in. If any request fails for the predefined threshold, it stops sending requests outside and responds with a predefined response or method. It will initiate the timer to maintain the open state of the service.

Spring also provides support for Netflix Hystrix. One can use this facility by adding some annotation, and it is done. Consider an online movie ticket booking site. If the booking of one ticket failed or responds in time out (due to heavy traffic), we want to recommend another movie showing at that time to the users so that they can book tickets for that.

There will be a ticket booking application and there be a calling application, say consuming application. The consuming application will be calling the exposed URL `http://<application Ip:port>/bookingapplication/{userId}/{movieId}` from the ticket booking application. In the case of this URL failure, we can configure a fall back method with some simple annotation. This will be the code-consuming application:

```
package com.practicalMicroservice.booking;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
@SpringBootApplication
@EnableCircuitBreaker

public class ConsumingApplication {

  @Autowired
  private ConsumingService consumingService;

  @RequestMapping(method = RequestMethod.POST,value =
"/book/{movieId}",produces = "application/json")
  public String book ticket(@PathVariable("userId") String
userId,@PathVariable("movieId") String movieId){
    return  consumingService.bookAndRespond(userId,movieId);
  }

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

The `@EnableCircuitBreaker` annotation enables the use of the circuit breaker pattern in the Spring Boot application. To use Hytrix in your application as the circuit breaker pattern, there is another annotation `@HystrixCommand`. However, it can be used only in classes that are annotated with `@service` or `@component`. So, we will create a service class here:

```
@Service
public class ConsumingService {

  @Autowired
   private  RestTemplate restTemplate;

 @HystrixCommand(fallbackMethod = "getAnotherCurentlyShowingMovie")
  public String bookAndRespond() {
```

```
    URI uri = URI.create("http://<application
Ip:port>/bookingapplication/{userId}/{movieId}");

    return this.restTemplate.getForObject(uri, String.class);
  }

  public String getAnotherCurentlyShwoingMovie() {
    return "We are experiencing heavy load on booking of your movie. There
are some other movies are running on same time, please check if you are
interested in any one. " + getSameShowtimeMovieList() ;
  }

  public String getSameShowtimeMovieList() {
       return "fast and Furious 8, The Wolverine3";
    }
}
```

In the preceding class, whenever the circuit fails, it will not call the infected service anymore; rather it will call another method `getAnotherCurentlyShowingMovie` and show the user another running movie at the same time.

The circuit breaker pattern is good to handle failure in synchronous communication, but it doesn't resolve other issues in synchronous communication. Knowing the existence of other services enables coupling between services, and too much coupling converts the microservice to a monolithic application.

Asynchronous communication

In asynchronous communication, unlike the synchronous service, there is no waiting for a response from the called service. In this pattern of communication, the calling service publishes its request and continues the rest of the work. In the same way, the calling service also has one listener inside itself, which will listen to a request generated by other services for this service. It captures those requests and processes it. The main benefit of this pattern is that it is non-blocking and does not wait for a response. Normally, the message broker is used for communication in this kind of pattern. The calling service is just interested in the message being delivered until the message is broken, and then it continues to its work. This way, even if the called service is not up, it doesn't affect the calling service. It makes the service more decoupled, and it need not know of the existence of the other service.

This kind of pattern basically works on the assumption that in one service, during the processing of its own request cycle, it doesn't need some data from another service. Rather it triggers the other service after completing its own request cycle. If the service needs data, which is with some other service, it replicates it in its own context by receiving a message from that domain. We define eventual consistency as follows.

Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

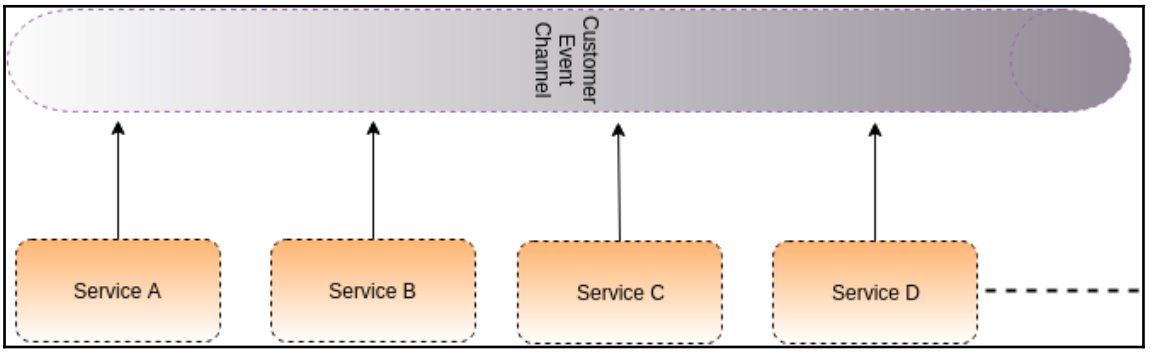
Message-based/event-based asynchronous communication

The main concept of asynchronous communication is basically messages flowing from one peer to another. These messages can be some command or some events, driven information. Asynchronous communication is typically of two types:

- Message based
- Event based

Both types work in the same manner. A message is triggered from the called service after finishing its work which lets the other services do their own work. There is no well-defined difference between these communication patterns, but message-based communication works on peer-to-peer communication, and event-based communication works based on publisher/subscriber. In message-driven communication, cohesion is obtained with a predefined queue name or channel. In the case of event-driven or command-based communication, the predefined format has to be shared. As shown in the following diagram, **Service A** is pushing the message to a predefined queue, which means logically, **Service A** knows of the existence of **Service B** as it knows that **Service B** will listen to the message and take further action on it:

On the other hand, in event-based communication, **Service A** will raise an event and then forget about it. Maybe, **Service B** uses it or it may be **Service D** or any other new service which is added very recently. There could be a case when no service is reading it at all. So, there is a slight change in mindset. As shown in the following diagram, **Service A** finishes its work and publishes it without knowing whether it is useful for anyone or not. However, in the prior method, **Service A** knows **Service B** will listen to it. There could be data in the message sent by **Service A**, in a format which only **Service B** can read or which is useful only to **Service B**. In event-based communication, there is a general format and data of finished work, as it can be read by any old or new service. So, the format of the message or event should be standard across the microservice. The following diagram describe the scenario for event driven communication:



As both share common properties, from now onwards, to explain the characteristics of these two types of communication combined, we will refer to them as message-driven communication. The following are the points that describe the pros of message-driven communication:

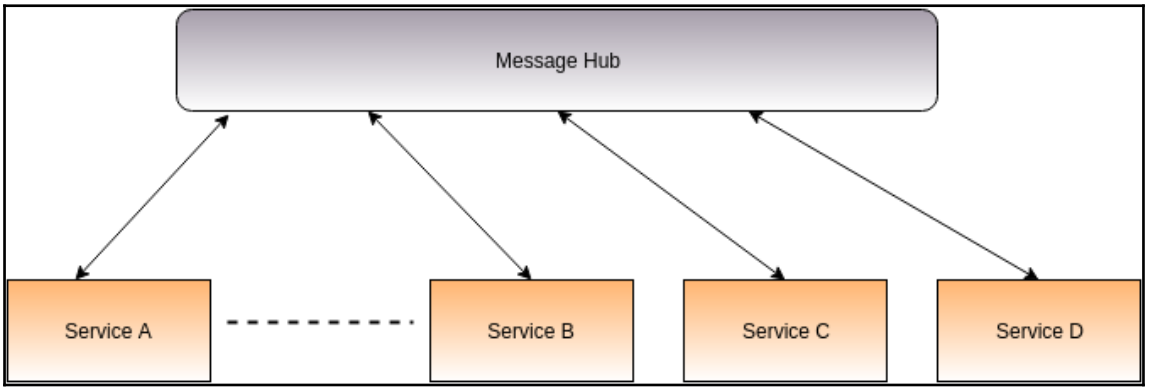
- As services are decoupled in asynchronous communication, easy scaling is the first benefit one can get from this style. Newly developed services can be added silently to receive messages/events and act on it. The sender doesn't need to know about the new receiver.
- In message-driven communication, back pressure can be handled in a better way than with synchronous communication . If the producer is creating the message more quickly than the consumer is consuming it, then there will be lots of messages in the queue to process. This situation is called back pressure. In this particular situation, the consumer can do two things. First, they can raise an event to tell the producer to slow down. Second, they can add more workers silently. This kind of problem mostly comes in event processing.

Here are the disadvantages of asynchronous communication:

- Operation complexity will increase, because message broker has to be installed, configured, and maintained with high availability, which is an overhead.
- It's hard to see what is happening in a system. There are messages flowing and action is being taken by different services, but due to the autonomous nature, it is hard to get a clear view of what's happening in the system. Due to this, extra effort is required at the time of debugging any issue.
- Sometimes, it is hard to convert any request/response structure to message-based communication. Consider a train booking system. Without checking the availability at run time, booking a ticket is not possible. In these kind of scenarios, synchronous communication makes sense.

Implementing asynchronous communication with REST

Asynchronous communication can happen with REST full service as well as any message broker. In a REST-based asynchronous communication, one can have a message handler hub as a microservice. Every microservice pushes its message to the hub, and now, the hub is responsible for passing the message to the respective service. In this way, the calling service waits to submit the message. It does not wait for the response coming from the called service. The following diagram showcases this concept:

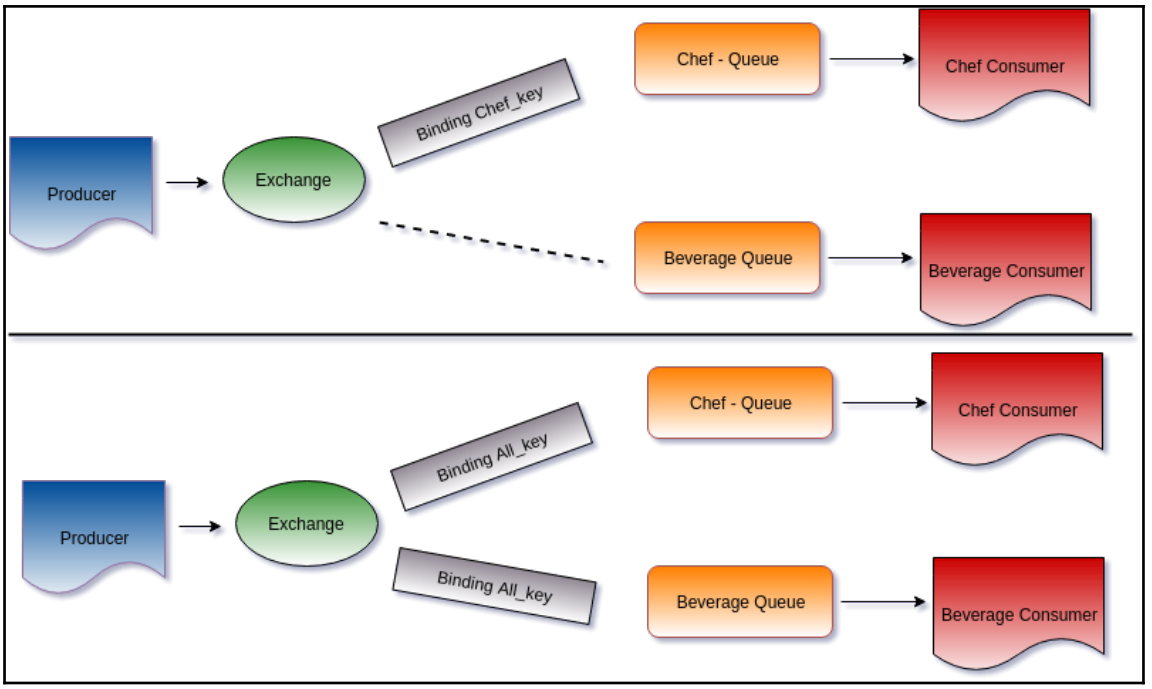


As shown in the preceding diagram, **Service A** finishes its work and then submits a message to the message hub. There could be two ways the message hub can pass this message to other services. One way is to make the message hub intelligent and read the packet and send this packet to the relevant microservice. The second way is that it will send the message to every service registered to the message hub. In this case, the relevant service will take action while other services will discard the message.

Implementing asynchronous communication with message broker

For implementation, we are taking an example of a burger shop, say "Crispy Bun." It is a drive-through burger shop where customers place their orders at one window and wait for the orders are the next window in their cars. There is a system that takes an order at the first window and pushes the order as a message or event/message to some queue/topic. There is another component in front of the chef where all the orders are listed. So, our requirement from this is that the order should be able to submit some middle broker (queue/topic) so that all the chefs can listen. The order should not vanish if no chef component is available, and it should save orders until the chef component reads it for processing. This example can be expanded. For example, the number of components can listen to this topic, or the same order should be passed to the chef component and the packing team as well so that they can start making boxes to pack this order, or perhaps it should be passed to the beverage team to make a beverage (if it is in the order). So, different components can listen to this order and can take respective actions for it. For simplicity, we are taking one order-taking component and one chef component.

RabbitMQ is one of most popular message broker solutions with the implementation of **Advance Message Queing Protocol** (**AMQP**). It is based on the Erlang language. It is different from a normal message broker. Here, messages are published directly to the queue rather than the producer sending messages to an exchange. Exchanges are messages responsible for messages routing to different queues based on attributes, binding, and routing queues. Many queues can be on the same exchange with a different routing key. For instance, in our example, there could be a beverage placed in the order. Then, the producer can send a message for the order created with either the **Binding Chef_key** binding key or with the **Binding All_key** key. The message with **Binding All_key** will go to both queues, but the message with the binding **Binding Chef_key** will only go to **Chef - Queue**. If any order doesn't have beverages included, then the message will go with **Binding Chef_key**. The following diagram describes the concept:



It is highly recommended that you have knowledge of RabbitMQ. Here is a brief description of how to install RabbitMQ on a Unix-flavored machine:

1. Run the `sudo apt-get update` command.
2. Add RabbitMQ application repository and key using the following commands:

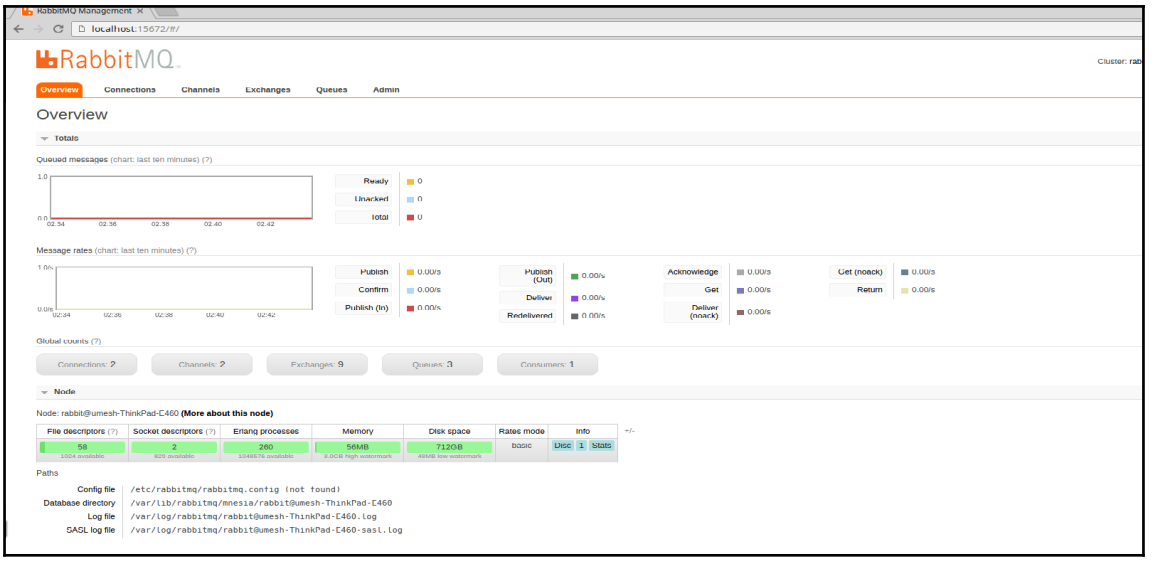
```
echo "deb http://www.rabbitmq.com/debian/ testing main" >>
/etc/apt/sources.list
curl http://www.rabbitmq.com/rabbitmq-signing-key-public.asc |
sudo apt-key add -
```

3. Update again, using the `sudo apt-get update` command.
4. Then, run the `sdo apt-get install rabbitmq-server` command.

It will install Erlang and RabbitMQ on the machine and start the RabbitMQ service. In case the service is not started automatically, you can use the `service rabbitmq-server start` command to start the RabbitMQ. Along with this installation, you might also need the admin console for handling the cluster and nodes of RabbitMQ. To do this, you need to install RabbitMQ plugins using the following command:

```
sudo rabbitmq-plugins enable rabbitmq_management
```

Once it is done, you can open up the management console on `http://<yourip>:15672`. It will show a page similar to the following screenshot. The default user name and password to open this is `guest`:



As we mentioned earlier, there will be two applications: one is the producer and one is the consumer. Now that our RabbitMQ is up, we will write a small code for the producer that will actually produce the order and submit it to the default exchange. The default exchange has no name. It is an empty string `""` and the message will be forwarded to the queue named `crispyBunOrder`.

Its `pom.xml` file will be as follows:

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

    <groupId>com.practicalMircorservice</groupId>
    <artifactId>EventProducer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>EventProducer</name>
    <description>com.practicalMircorservice </description>
```

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.1.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-
        8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
        8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
          <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>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Camden.SR1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

In the producer application, there will be two Java files:

- `EventProducerApplication.java`: This file will be the main application and will have the REST controller in it
- `CrispyBunOrder.java`: This is the order object that will be submitted

The following is the code for `EventProducerApplication.java`:

```
@SpringBootApplication
@EnableBinding
@RestController
public class EventProducerApplication {

    private final String Queue = "crispyBunOrder";
    @Autowired
    private RabbitTemplate rabbitTemplate;

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

    @RequestMapping(method = RequestMethod.POST, value =
    "/orders/{orderId}")
    public void placeOrder(@PathVariable("orderId") UUID
    orderId,@RequestParam("itemId") Integer
    itemId,@RequestParam("userName") String userName) {
        CrispyBunOrder orderObject =
        createOrder(orderId,itemId,userName);
        rabbitTemplate.convertAndSend(Queue,orderObject);
    }

    private CrispyBunOrder createOrder(UUID orderId,Integer itemId,
    String userName){
        CrispyBunOrder order = new CrispyBunOrder();
        order.setItemId(itemId);
        order.setOrderId(orderId);
        order.setUserName(userName);
        order.setOrderPlacedTime(new Date());
        return order;
    }
}
```

The preceding class will take the `orderId`, `itemId`, and `userName` as a parameter and create an order. It will submit the order to the queue named crispy bun order. As we have added cloud stream RabbitMQ dependencies in the **Project Object Model** (**POM**) file, Spring Boot will create a Rabbit template automatically for us. With the help of the RabbitMQ template, we can send any object to the given queue name. Actually here, we are submitting to the default exchange. The default exchange has no name. It is an empty string `""`. So, any message submitted to the default exchange will go directly to the queue name.

The `CrispyBunOrder` object class will have four parameters and will look like this:

```
package com.practicalMircorservices.eventProducer;

import java.io.Serializable;
import java.util.Date;
import java.util.UUID;

public class CrispyBunOrder implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 6572547218488352566L;

    private UUID orderId;
    private Integer itemId;
    private Date orderPlacedTime;
    private String userName;
    public UUID getOrderId() {
        return orderId;
    }
    public void setOrderId(UUID orderId) {
        this.orderId = orderId;
    }
    public Integer getItemId() {
        return itemId;
    }
    public void setItemId(Integer itemId) {
        this.itemId = itemId;
    }
    public Date getOrderPlacedTime() {
        return orderPlacedTime;
    }
    public void setOrderPlacedTime(Date orderPlacedTime) {
        this.orderPlacedTime = orderPlacedTime;
    }
    public String getUserName() {
        return userName;
```

```
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

Its `application.properties` will have two properties:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
```

These are the default ports of RabbitMQ.

Now, coming to the consumer side, which is an altogether new application named `EventConsumerApplication`. This application can be created by downloading the generated code from `start.spring.io`. Before downloading make sure you have clicked the checkbox of **Stream Rabbit**. In this application, the `CrispyBunOrder` class will be the same as the producer, as we have to deserialize it here. Other than that, there will be one listener, who will be listening to the RabbitMQ queue name `crispyBunOrder`. The code of this class will be as follows:

```
package com.practicalMircorservices.eventProducer;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.messaging.handler.annotation.Payload;



@SpringBootApplication
@RabbitListener(queues = "crispyBunOrder")
@EnableAutoConfiguration
public class EventConsumerApplication {

    @Bean
    public Queue crispyBunOrderQueue() {
        return new Queue("crispyBunOrder");
    }
    @RabbitHandler
    public void process(@Payload CrispyBunOrder order) {
        StringBuffer SB = new StringBuffer();
        SB.append("New Order Received : \n");
```

```
            SB.append("OrderId : " + order.getOrderId());
            SB.append("\nItemId : " + order.getItemId());
            SB.append("\nUserName : " + order.getUserName());
            SB.append("\nDate : " + order.getOrderPlacedTime());
            System.out.println(SB.toString());
        }

        public static void main(String[] args) throws Exception {
            SpringApplication.run(EventConsumerApplication.class, args);
        }

    }
```

The `@RabbitListener(queues = "crispyBunOrder")` annotation defines which queue it has to listen to. In addition to this, we can define many parameters here, with listeners such as exchange name, routing key, and so on. In that case, the annotation will be as follows:

```
    @RabbitListener(bindings = @QueueBinding(
            value = @Queue(value = "myQueue", durable = "true"),
            exchange = @Exchange(value = "auto.exch"),
            key = "orderRoutingKey")
      )
```

Start both the application/components in different consoles with `mvn spring-boot:run`. Make sure both run on different ports by adding the `server.port` property in `application.properties`.

Now, you can use the curl command from the command line to hot the producer URL to test the response , like `curl -H "Content-Type: application/x-www-form-urlencoded" --data "itemId=1&userName=john" http://localhost:8080/orders/02b425c0-da2b-445d-8726-3cf4dcf4326d`; it will result in the order showing on the consumer console. This is the basic understanding of how messaging works. There are more complex examples of this in real life such as taking user action feed from a website, stock price feed, and so on. Mostly, it comes in handy when you have a stream of data coming, and based on the routing key, the broker can send the data to different queues. Kafka is another good tool for doing this. Spring has inbuilt support for Kafka, RabbitMQ, and some other messaging brokers. This inbuilt support helps developers to setup and start development with supported messaging brokers pretty quickly.

Financial services

Extending our example of a credit scoring application, we have to add a new microservice named `FinancialServices`. In `Chapter 2`, *Defining Microservice Components*, we created a `UserServices` microservice. That service is suppose to store the user's personal details along with addresses. In this chapter, we will learn about financial microservices which are supposed to store the financial details of users, which include bank account details and the user's financial obligations. The table structure of this microservice is as follows:

```
CREATE TABLE `bank_account_details` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
   `user_id` char(36) NOT NULL,
  `bank_name` varchar(100) DEFAULT NULL,
  `account_number` varchar(20) NOT NULL,
  `account_holders_name` varchar(250) NOT NULL,
  `fsc` varchar(20) DEFAULT NULL,
  `account_type` int(11) DEFAULT NULL,
  `created_on` datetime(6) NOT NULL,
  `deleted_on` datetime(6) DEFAULT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1;

CREATE TABLE `obligations_details` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` char(36) NOT NULL,
  `total_monthly_spending` decimal(10,0) NOT NULL,
  `monthly_income_supports` decimal(10,0) NOT NULL,
  `monthly_emi_payout` decimal(10,0) NOT NULL,
  `created_on` datetime(6) NOT NULL,
  `deleted_on` datetime(6) DEFAULT NULL,
  PRIMARY KEY (`Id`),
  KEY `Obligations_user_id` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Boot strap properties files will contain two properties in it:

```
spring.application.name=financialService
spring.cloud.config.uri=http://localhost:8888
```

Also we have to create `financialService.properties` inside the `configServer` code at location:

```
server.port=8090
Connect.database=financialDetails
spring.datasource.url=jdbc:mysql://localhost:3306/financialDetails
spring.datasource.username=xxxxxx
spring.datasource.password=xxxxxxxx
```

```
# optional Properties
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL5Dialect
spring.jpa.hibernate.naming_strategy=org.hibernate.cfg.ImprovedNamingStrate
gy
```

The creation of this application is the same as the `UserService` application. You have to download the templated code from `start.spring.io`. Use the application name as `FinancialServiceApplication` while creating a project on `start.spring.io` and hit the generate button and download the code. The main application file will look pretty much similar to the user service application:

```
package com.practicalMicroservcies;

import org.flywaydb.core.Flyway;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class FinancialServiceApplication {
    @Value("${spring.datasource.url}")
    private String url;
    @Value("${spring.datasource.username}")
    private String userName;
    @Value("${spring.datasource.password}")
    private String password;
    @Value("${Connect.database}")
    private String database;

    @Bean(initMethod = "migrate")
    public Flyway flyway() {
        String urlWithoutDatabaseName=
        url.substring(0,url.lastIndexOf("/"));
        Flyway flyway = new Flyway();
        flyway.setDataSource(urlWithoutDatabaseName, userName,
        password);
        flyway.setSchemas(database);
        flyway.setBaselineOnMigrate(true);
        return flyway;
    }

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

```
    }
}
```

As we have two tables in this, we will have two entity tables: bank account details and obligation details. The code in these files will look like this:

```
@Entity
@Table(name = "bank_account_details")
public class BankAccountDetail implements Serializable {

    private static final long serialVersionUID = 4804278804874617196L;



    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    private int id;

    /**
     * User Id , unique for every user
     */

    @Column(name = "user_id", nullable = false, unique = true, length =
    36)
    private String userId;

    /**
     * Bank Name of User
     */
    @Column(name = "bank_name", nullable = false, length = 100)
    private String bankName;

    /**
     * Account number of User
     */
    @Column(name = "account_number", nullable = false, unique = true,
    length = 20)
    private String accountNumber;

    /**
     * Account holder Name of account
     */
    @Column(name = "account_holders_name", nullable = false, length =
    250)
    private String accountHolderName;

    /**
```

```
     * Any National Financial system code of bank
     */
    @Column(name = "fsc", nullable = false, length = 20)
    private String fsc;

    /**
     * Account Type of user
     */
    @Column(name = "account_type", nullable = false)
    private Integer accountType;

    /**
     * Date on which Entry is created
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_on", columnDefinition = "TIMESTAMP DEFAULT
    CURRENT_TIMESTAMP")
    private Date createdOn = new Date();

    /**
     * Date on which Entry is deleted.
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "deleted_on", columnDefinition = "TIMESTAMP DEFAULT
    NULL")
    private Date deletedOn;
}
```

Next, the entity class, obligation entity, which should be storing the user's financial obligation, and will be look like the following:

```
@Entity
@Table(name = "obligations_details")
public class ObligationDetails implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = -7123033147694699766L;

    /**
     * unique Id for each record.
     */

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    private int id;
```

```
    /**
     * User Id , unique for every user
     */

    @Column(name = "user_id", nullable = false, unique = true, length =
    36)
    private String userId;

    /**
     * Totally monthly income , may be from himself or wife (combined)
     */
    @Column(name = "monthly_income_supports", nullable = false)
    private BigDecimal monthlyIncome;

    /**
     * Monthly Emi to payout
     */
    @Column(name = "monthly_emi_payout", nullable = false)
    private BigDecimal monthlyemi;
    /**
     * totally month spending
     */
    @Column(name = "total_monthly_spending", nullable = false)
    private BigDecimal monthlySpending;

    /**
     * Date on which Entry is created
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_on", columnDefinition = "TIMESTAMP DEFAULT
    CURRENT_TIMESTAMP")
    private Date createdOn = new Date();;

    /**
     * Date on which Entry is deleted.
     */
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "deleted_on", columnDefinition = "TIMESTAMP DEFAULT
    NULL")
    private Date deletedOn;
}
```

It should include setter and getter methods for class properties.

And can override the `toString` method:

```
    @Override
    public String toString() {
```

```
        return "ObligationDetails [userId=" + userId + ",
        monthlyIncome=" + monthlyIncome + ", monthlyemi=" + monthlyemi
        + ", monthlySpending=" + monthlySpending + ", createdOn=" +
        createdOn + ", deletedOn=" + deletedOn + "]";
    }
```

In our user service, we will have repositories for both the entities (bank account details and obligation details):

```
package com.practicalMicroservcies.repo;
import org.springframework.data.jpa.repository.JpaRepository;

import com.practicalMicroservcies.entity.BankAccountDetail;

public interface BankAccountDetailRepository extends
JpaRepository<BankAccountDetail, Integer> {
    BankAccountDetail findByUserId(String userId);
}
```

And obligation repositories:

```
package com.practicalMicroservcies.repo;

import org.springframework.data.jpa.repository.JpaRepository;

import com.practicalMicroservcies.entity.BankAccountDetail;
import com.practicalMicroservcies.entity.ObligationDetails;

public interface ObligationRepository extends
JpaRepository<ObligationDetails, Integer> {
    ObligationDetails findByUserId(String userId);
}
```

There will be one service class named `FinancialServices` to handle queries from both repositories:

```
@Service
@Transactional
public class FinancialServices {

    @Resource
    BankAccountDetailRepository accountDetailRepo;

    @Resource
    ObligationRepository obligationRepo;
```

The following method is used to store the bank account details of the user:

```
    public void saveAccountDetail(BankAccountDetail accountDetail) {
        accountDetailRepo.save(accountDetail);
        System.out.println("AccountDetails Saved!");

    }

    public void saveObligation(ObligationDetails obligationDetails) {
        obligationRepo.save(obligationDetails);
        System.out.println("Obligation Details Saved!");

    }
```

We need a method to fetch the bank account details and obligations from the database based on the `userId` provided. The following are two methods in the service class, to do the get data job for us:

```
    public ObligationDetails getObligationDetail(UUID userId) {
        ObligationDetails returnAddressObject =
        obligationRepo.findByUserId(userId.toString());
        return returnAddressObject;

     }

    public BankAccountDetail getAccountDetail(UUID userId) {
        BankAccountDetail userObjectToRetrun =
        accountDetailRepo.findByUserId(userId.toString());
        return userObjectToRetrun;

    }
```

The following is the method to delete the financial details of the user. This method doesn't actually delete the data from the database. Rather it just marks the data as deleted. Any data destruction statement should be avoided for the database as much as possible.

```
    public void deleteFinancialDetail(UUID userId) {
        BankAccountDetail accountObject =
        accountDetailRepo.findByUserId(userId.toString());
        accountObject.setDeletedOn(new Date());
        accountDetailRepo.saveAndFlush(accountObject);
        ObligationDetails obligationObject =
        obligationRepo.findByUserId(userId.toString());
        obligationObject.setDeletedOn(new Date());
        obligationRepo.saveAndFlush(obligationObject);
    }

}
```

The controller will look like as follows:

```
@RestController
@RequestMapping("/PM/finance/")
public class FinancialController {
    private static final Logger logger =
Logger.getLogger(FinancialController.class);

    @Resource
    FinancialServices financialService;

    @Resource
    ObjectMapper mapper;
    /**
     * Method is responsible for adding new AccountDetails.
     *
     * @param address
     * @param userId
     * @return
     */
    public static final String addAccountDetails =
    "addAccountDetails(): ";

    @RequestMapping(method = RequestMethod.POST, value = "
    {userId}/account", produces = "application/json", consumes =
    "application/json")
    public ResponseEntity<String> addAccountDetails(@RequestBody
    BankAccountDetail accountDetail, @PathVariable("userId") UUID
    userId) {
        logger.debug(addAccountDetails + " Account for user Id " +
        userId + " is creating.");
        accountDetail.setUserId(userId.toString());
        financialService.saveAccountDetail(accountDetail);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    /**
     * Method is responsible for creating a obligation Details.
     *
     * @param userDetail
     * @param userId
     * @return
     */

    public static final String addObligationDetails =
    "addObligationDetails(): ";

    @RequestMapping(method = RequestMethod.POST, value = "
```

```
{userId}/obligation", produces = "application/json", consumes =
"application/json")
public ResponseEntity<String> addObligationDetails(@RequestBody
ObligationDetails obligationDetails, @PathVariable("userId") UUID
userId) {
    logger.debug(addObligationDetails + " Creating user's
    obligation with Id " + userId + " and details : " +
    obligationDetails);
    obligationDetails.setUserId(userId.toString());
    financialService.saveObligation(obligationDetails);
    return new ResponseEntity<>(HttpStatus.CREATED);
}

/**
 *  Deleting Financial Detail of user
 * @param userDetail
 * @param userId
 * @return
 */

public static final String deleteFinancialDetails =
"deleteFinancialDetails(): ";

@RequestMapping(method = RequestMethod.DELETE, value = "{userId}",
produces = "application/json", consumes = "application/json")
public ResponseEntity<String> deleteFinancialDetails(
@PathVariable("userId") UUID userId) {
    logger.debug(deleteFinancialDetails + " deleting user with Id "
    + userId);
    financialService.deleteFinancialDetail(userId);
    return new ResponseEntity<>(HttpStatus.CREATED);
}

/**
* Method is responsible for getting the account detail for given ID.
*
* @param userId
* @return
*/
public static final String getAccountDetails =
"getAccountDetails(): ";

@RequestMapping(method = RequestMethod.GET, value = "
{userId}/account", produces = "application/json", consumes =
"application/json")
public ResponseEntity<BankAccountDetail>
getAccountDetails(@PathVariable("userId") UUID userId) {
    logger.debug(getAccountDetails + " getting information for
```

```
            userId " + userId);
            BankAccountDetail objectToReturn =
            financialService.getAccountDetail(userId);
            if (objectToReturn == null)
                return new ResponseEntity<>(HttpStatus.NOT_FOUND);
            else
                return new ResponseEntity<>(objectToReturn, HttpStatus.OK);
        }

        /**
         * Method is responsible getting the Obligation Detail.
         * @param userId
         * @return
         */
        public static final String getObligationDetails =
        "getObligationDetails(): ";

        @RequestMapping(method = RequestMethod.GET, value = "
        {userId}/obligation", produces = "application/json", consumes =
        "application/json")
        public ResponseEntity<ObligationDetails>
        getObligationDetails(@PathVariable("userId") UUID userId) {
            logger.debug(getObligationDetails + " getting Obligation
            Details for user Id: " + userId);
            ObligationDetails objectToReturn =
            financialService.getObligationDetail(userId);
            if (objectToReturn == null)
                return new ResponseEntity<>(HttpStatus.NOT_FOUND);
            else
                return new ResponseEntity<>(objectToReturn, HttpStatus.OK);
        }
    }
```

This completes the code of financial microservices. The endpoint URL for this will be as follows.

For creating the obligation, the endpoint will be:

```
POST  http://localhost:8090/PM/finance/<user Id>/obligation
```

An example of this URL will be as follows:

```
POST
http://localhost:8090/PM/finance/93a52ce4-9331-43b5-8b56-09bd62cb0444/oblig
ation
```

For creating the bank account details, the endpoint will be:

```
POST http://localhost:8090/PM/finance/<user Id>account
```

An example of this URL will be as follows:

```
POST
http://localhost:8090/PM/finance/93a52ce4-9331-43b5-8b56-09bd62cb0444/accou
nt
```

If we make the `GET` request of the same endpoint, then it will fetch the details of the user from the database and send it back as JSON. For instance if we want to know the obligation for any user, hit the `GET` request to the endpoint mentioned above like:

```
GET
http://localhost:8090/PM/finance/93a52ce4-9331-43b5-8b56-09bd62cb0444/oblig
ation
```

The response would be a JSON something like the following service:

```
{
"id": 1,
"user_id":"3a52ce4-9331-43b5-8b56-09bd62cb0444",
"monthlyIncome":150000,
"monthlyemi":50000,
"monthlySpending":50000,
"createdOn":"1483209000000",
"deletedOn":null
}
```

This service helps store the financial information of users. It consists of two tables: one to store the bank detail information of the users and the second table to store monthly income, household spending, monthly EMI spending, and so on. All these decisions will help us in creating the credit scoring microservice. In the same way, there is another microservice called employment microservice. This stores information regarding employment details such as salaried or self-employed, who is the current employer, and the employee's current salary. The implementation of this service is very similar to what we already have, so I'm leaving it to the readers.

Summary

In this chapter, we discussed the principles of communication between microservices and various methods of communication such as synchronous or asynchronous, and communication patterns such as choreography and orchestration. We have seen the example of event-based messaging and discussed various use cases and ways of doing it. REST can be used in both types of communication: synchronous and asynchronous. There are some tools to handle both type of communication. It entirely depends on the situation and the needs of the specific project to choose one pattern over the other. In the next chapter, we will discuss how to secure the microservices with an example of our credit-scoring microservice.

4
Securing Microservice Endpoints

In the last chapter, you learned about various ways of communication between microservices. Communication can be either of the two patterns, synchronous and asynchronous. This discussion leads us to a very interesting question, "How can one manage the security during a communication between microservices or with outer worlds?" Each service in the microservice architecture is defined to perform a set of functions. One more responsibility would be security communication, which is against the definition of microservices. Security is a major concern here in terms of authentication and authorization. In this chapter, we will dig more into the details of security of microservices.

In this chapter, we will cover the following topics:

- Security challenges in microservices
- Using **JSON Web Token** (**JWT**) along with OpenID and OAuth 2.0
- How JWT can implemented in a sample application
- Development of the credit scoring microservice

Security challenges in microservices

We are in a fast-paced industry. New technologies are entering the industry fast, and security concerns are also increasing with the same speed. Security is no one of the ultimate concerns in any software architecture. In a monolithic application, less surface areas are exposed to the outer world, but once it is breached with any single point, there is a high chance that the attacker can reach the entire system. As monolithic architecture has been around for a while now, there are some attack patterns that have been followed, and standards are developed to prevent them. Some famous types of attacks are spoofing, tampering, **denial-of-service** (**DoS**), escalation or privilege, information disclosure and so on.

The attacks mentioned earlier can happen in microservices also. One benefit we have in microservices is that unlike monolithic architecture, if an attacker gets into one microservice, it does not allow them to infect the other microservices. This is because each microservice has its own security layer. Although microservice architecture is different from monolithic architecture, to prevent microservice from attacks, a strategy similar to monolithic architecture can be applied. Let's discuss some of the challenges we face in a microservice environment and some security practices.

Mix technology stack or legacy code presence

In microservices, one has the leverage of having a heterogeneous stack of technology. Now, if there is diverse technology used in the microservice, then it would require the effort of using any generic security layer. Perhaps implementation in different languages has its own complexity. The complexity increases if it has some legacy part also. Then, the developer has to match the consistency with old code. Legacy code might have some limitations, which sadly leads to some compromises in the new security layer.

Authentication and authorization (access control)

These are the fundamental concepts in the security context. In plain language, we can describe it as authentication. Authentication asks the question, "Who are you?" and authorization says "What you can do?" Logging in to any system is a process of authentication. Assigning a role to a user at the time of login whether they are an admin or a normal user is a process of authorization. Access control should be there. Each entity should have minimum access, which is required to do business.

Token-based security

If token-based security is chosen by the developer, then the choice of token format is a challenge. The token should be well managed. Its life cycle and expiry time should be taken into consideration to understand whether the token is stored on some device where it can be hacked later and used for different purposes. JWT here seems to be the best candidate.

Responsibility of security

In general, a developer comes with a mindset that security is the job of the network or deployment's team. However, in a microservice environment, each service is considered to be open to threats. Of course, the network layer will have its own kind of security, such as port blocking, IP white listing, and so on. However, it becomes a challenge for the organization to change the developer's mindset to think about security while coding, to log any suspicious activity.

Fear with orchestration style

Orchestration pattern in a microservices architecture has a different kind of threat. In orchestration, one component rules all components. So, if someone breaches or attacks this layer, they can damage or take control of all the components' behavior. So, this one component rules all concept is always feared.

Communication among services

There is a lot of data flow between services in a microservice architecture. Making that data tamper proof is another challenge; how calling service ensures that data received by it, is not tampered with in between communications. Also, there should be some identifier to know which data is sensitive, hence extra care needs to be taken; for example in cases of payment data.

We just hit the tip of the iceberg. There are many challenges faced in the security of the microservice. The API Gateway can also be a solution to some of these threats. Fortunately, we have some standard practices that can be followed and assist us with the basics of security.

There are some common practices for security that should be considered in microservice architecture:

- The basic thing is to use certificates during communication, even HTTPS will be a good start here (if communication is REST based).
- Encrypt traffic among services communication.
- Make the correct access strategy. Access between components should be restricted. It should be started with the lowest permission and given only the required permission.
- Monitoring is another tool to deal with the security aspect. If good monitoring is in place, it can give you a good view of what's happening in the system.
- Log all the things required to debug or identify any security issues. It will be helpful for the developer's team as well.
- Test your security infrastructure on regular basis.
- If any third-party code or container is used in your microservice release, you should verify or scan that completely before using. Using this kind of stuffmeans you need to keep eyes on any security patch updates.
- **Open Web Application Security Project** (**OWASP**) has some very nice tips and cheat sheets for security. These should be considered while thinking in terms of security.
- Code review should be done very frequently and with less lines of code. If you ask someone to do a code review of more than 1000 lines, you will hardly get any comments. In contrast, if you share less than 100 lines of code, you will get more comments.

Normally, a token-based system security is adapted from the security designed in the microservice architecture. Its strategy is not only for REST-based requests, but can be used in other formats also. The concept is based on the idea that the client gets the token first and embeds that token in each subsequent request. This token will be issued to the client after authentication and authorization from the authentication service. The resource owner will verify this token with the token generation server and check whether this token is valid to serve this request. It will work this way until the token expires. Once the token is expired, then the client has to provide its credential to the token provider again and get the new token. JWT is the most commonly used format for REST-based microservices. JWT goes one step ahead and adds the scope and action permit to the client in the token and seals it with encryption algorithm. The next section discusses this authentication and authorization framework and JWT tokens in detail.

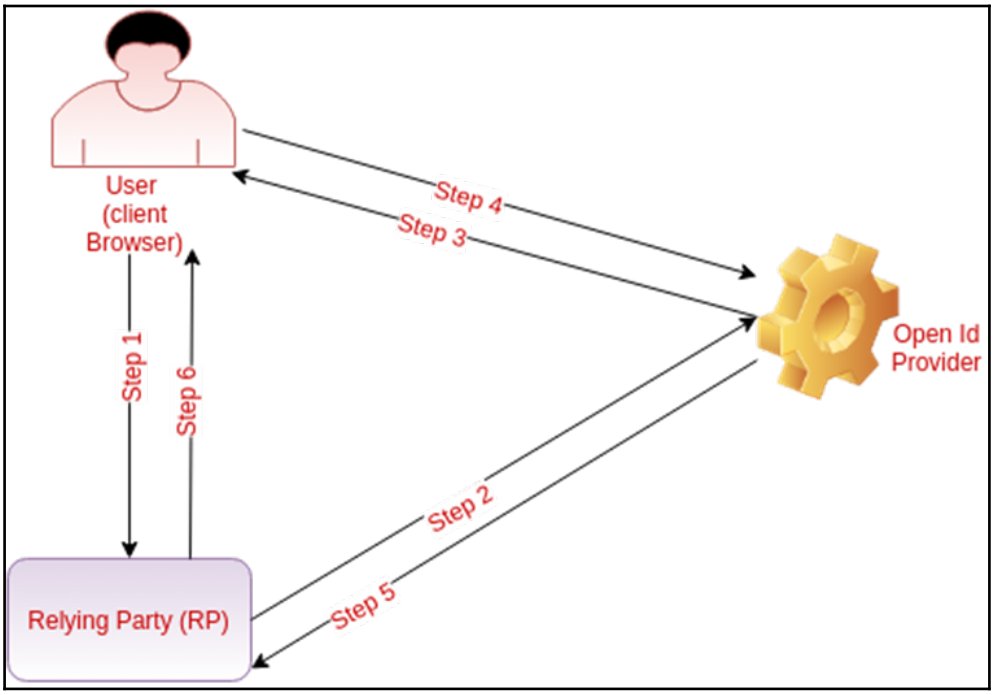
Using JWT along with OpenID and OAuth 2.0

JWT stands for JSON web token, which actually has some information related to a particular call. JWT is issued with both authentication and authorization. From a valid JWT token, we can easily identify who is the user and what they can do. Before understanding the structure of JWT, let's get familiar with two other terms: OpenID and OAuth. Nowadays, OAuth 2.0 and OpenID Connect are looked at as an alternative to **Security Assertion Markup Language** (**SAML**) for communicating identities and information about a user to identity providers and service or resource providers. OpenID is more for authentication purposes, and OAuth is more for the authorization server. The following section gives more details about these two terms.

OpenID

Jumping to a few years back, every website had its own login mechanism. The user had to maintain credentials for each website separately. The user might use a different email, website, different sites for shopping, traveling, booking, and so on. So, the problem users were facing was to keep track of all these credentials. This initiated the idea of **single sign-on** (**SSO**). Conceptually, the user has one username and password, and they can use different websites with the same credentials. It makes the sign-in process faster and easier. Here, OpenID comes into play. OpenID is a way to identify any particular user in all the websites they visit. With OpenID, the user will give the password to an OpenID provider, and the provider will tell the website (which the user has visited) who they are. In this way, the password will never be saved on the visited website.

There are many OpenID providers in the market. It depends on which one you have chosen. These days, you see a section on the login page with the **Log in with Google+** text. It implies that if you are a Google OpenID user, you can log in to a website using those credentials. In fact, if you have a Gmail/Google account, it means you are a Google OpenID user. Other famous sites like Facebook, Twitter, LinkedIn, and so on also provide the OpenId mechanism. When you use these kinds of OpenID providers, you will be redirected to the page where you will be entering login/password credentials for that OpenID account. In our example, it would be Gmail credential. The request will go to the Google OpenID server. It will authenticate your credentials and share the response with the website the user visited. In addition to the authentication result, it also shares some information regarding the user, such as their email and so on. Before sharing the information, the OpenID server takes consent from the user; the user can allow or disallow the website from seeing the information:



The preceding diagram briefly explains how a normal OpenID provider works. Here are the steps:

1. The user chooses the option login with the OpenID provider (say Google).
2. **The relying party** (**RP**) (website that the user visited) passes the request to the OpenID provider to authenticate the user and share information.
3. **OpenID Provider** (**OP**) asks for credentials from the user and permissions to share information with the mentioned third-party website/application.
4. If the user allows it, then the credentials come back to the OP, and it authenticates the credentials.
5. OP shares the login result and information with RP.
6. RP welcomes the user in its website/application.

This is the simple work flow of OpenID provider. So, no matter how many websites the user visited, if they are connected with the same OpenID provider, then the user has to remember only one username/password. This is the simplicity of the OpenID concept.

OAuth 2.0

One good thing we get from OpenID is that we can identify the user. For microservices, it changes the challenges faced. In microservices, each service is performing its own kind of work. Now, the challenge is, if the user or client is authenticated, should he get access to all the microservices? This will allow the user to mess with all the microservices, which means the whole system that makes this architecture is treated as monolithic architecture. Here, if the attacker breaches the security, then they can get access to the whole system.

This problem leads to another idea. Each user/client should have limited permissions. Not every user/client should have access to all the microservices and all the operations in a particular microservice. In this architecture, each service should be implemented in this way to check the permission authority before performing any operation. Each client/user has to present the claims to the called service; that claim has permission to perform such operation. OAuth is the concept that can help in implementing this scenario.

OAuth 2 is an authorization framework that enables applications to obtain limited access to resources on an HTTP service. OAuth was started when developers from Twitter and Gnolia met in 2006 and discussed how to authenticate internet services. They knew there was no standard for API access delegation at that time. They founded an OAuth committee on the internet in April 2007, and prepared and shared a draft of the OAuth proposal. Later, it was included in IETF.

OAuth 2 is different from the authentication or login processes. Although efforts are being made worldwide to make OAuth 2 much more efficient to use alone, for now, OpenID for authentication and OAuth 2 for authorization seem to be the combination accepted worldwide. Let's understand the use of both with real-life examples. In a school, a guest comes to meet a student. That guest is, say, a relative of the student. The steps for the guest to meet the student will be as follows:

1. The guest goes to the school reception and says that they want to meet the student.
2. A notification is sent to the student about the guest.
3. The student comes and identifies the guest as a relative.
4. Until now, it is OpenID that is working. When any service or user provides its credentials or claims to be someone, then OpenID verifies those credentials or claims.
5. Now, the guest will get a visitor tag.
6. Both the student and the guest can meet in a room, say, the parents room.

Now, this guest can pass the reception but has access to a limited area, such as the parents room or cafeteria, but not beyond that. That's what is defined by OAuth. OAuth is a delegation protocol and is used for authorization. OAuth can be used to authenticate the user also. Facebook is using it for authentication, but the primary focus of OAuth is for authorization.

OAuth mainly has four roles:

- Resource owner
- Resource server
- Authorization server
- Client

The resource owner is the user who authorized or provided credentials for access to the account. The resource server is the one who holds the resources used by RO. The authorization server is the one who authenticates those credentials and verifies the identity. A client can be a third-party application that wants access to the account.

The original idea is that one has to register the application with an OAuth provider server. The client ID and client secret are shared by the server. So, the flow here is like this:

- The user tries to access the resources of the client application; this could be a web page.
- Client redirects the user (RO) to the Auth server. The client includes its client ID and secret also in the request.
- The user (RO) authorizes the client application.
- The Auth server gives the authorization grant back to the client.
- On behalf of the user, the client asks for a token from the Auth server.
- The resource owner uses that token to give access to the resource to the user.

JWT

JWT is a special format of token to communicate between different services. As there are so many proprietary standards in the industry, but no universally accepted token standards, JWT fills that space. It makes the token format standard. Details of this format can be found in RFC documents at `https://tools.ietf.org/html/rfc7519`. It not only carries information about authentication, but also much more metadata in it. Issuer name and scope of permission are very common examples of these. It can also have some customized metadata.

OpenID and OAuth are the frameworks for authentication and authorization. Both of them talk about the concept of access information, but there is nothing mentioned about encryption and format of the token. So, JWT is just the specification of how this token should be wrapped and encrypted. One can have its own identity server and can have a separate format of token, but when it comes to standardization, then JWT is the most appropriate token format.

JWT has its own structure and a different section. JWT has the following three segments in it:

- Header section
- Payload section
- Signature section

All these sections are separated by "." in JWT token. Just for the sake of understanding these are:

- `EyJ0eXAiOiJKV1QiLCJhbGciOiJS` (// first section header)
- `e-KqaMUWvxjR1VJyI8hzbrGV65fTvHY` (// second section of payload starting with)
- `U3gNnYT1KcrA22OYsJyvhKhTEE_Ztx` (// third section of signature starting with)

There is no new line between each token as shown here. The concept here is to split the token into three different sections on the basis of position of "." (the dot). Each segment should be decrypted separately.

A few examples

If we decode the JWT token, the structure will be something like this:

Header

Lets understand some of the terms used in these sections:

alg

It is a string type parameter. It is used to define the algorithm used in JWT to encrypt the data, such as RS256.

typ

It is a string type parameter. It is used to tell the structure of the token. For now, its value will be JWT. It is an optional field for now. It could be used in future for other structure values.

kid

It's a string parameter and a hint to define which specific key should be used to validate the signature.

x5u

It is a string parameter used to point to the URL X.509 certificate public key to validate the signature.

x5t

It is again a string parameter that provides the SHA-256 digest of the DER encoding of the X.509 certificate. This is base64 encoded. The kid parameter can have the same value that is mentioned here:

```
{
  "typ": "JWT",

  "alg": "RS256",

  "x5t": "ywSXXXXEiJkEXXXXXXXXXXK9SK-k",

  "kid": "ywSXXXXEiJkExXXXXXXXXXK9SK-k"
}
```

Payload

In payload, we can customize fields, but there are some standard fields also. So, here is an explanation of some of the standard fields used in the payload:

iss

It refers to the identity that issued the JWT token.

aud

It refers to the values that are the intended recipient of this token.

sub

It refers to the subject of the token. It's an optional parameter.

exp

It refers to expiration time. It normally contains the Unix time after which this token will be no longer valid.

nbf

It refers to the not-before time. It means this section identifies the time at which the JWT will start to be accepted for processing, not before that.

iat

It refers to issued-at time. It normally contains the Unix time at which this token was issued.

These are some standard parameters used in a JWT token. Let's move to running an application by which we can make our application more secure:

```
{
  "iss": "http://YourIdentitySever.com",

  "aud": "http://YourIdentitySever.com/resources",

  "exp": 1481576528,

  "nbf": 1481572928,

  "client_id": "84802e3f-7fde-4eea-914e-b9e70bdfc26f",

    // you can have any custom data here ,

 "scope": "admin_services"
}
```

Sample application

Now we have an understanding of JWT token, we can either implement or own a library to decode the JWT token, or we can use the already available public library. In the given example, we are using Spring Boot with a Spring security feature. Here, we will create a custom filter and try to put that filter after `UsernamePasswordAuthenticationFilter`. This filter will extract the token from the header and run the various checks. It checks whether the JWT token is valid, and also checks the permission and issuer of the token. Any fails in the check will result in relevant messages. The token is supposed to be generated by any identity server. The token is also verified by the same identity server. In the presented example, we are not implementing any identity server in Java. The client-side code is there in this code.

For a sample application, we are using the library from bitbucket jose4j. To use this library, add the following dependency in the `pom.xml` file:

```
<!-- For JWT Filter -->
        <dependency>
            <groupId>org.bitbucket.b_c</groupId>
            <artifactId>jose4j</artifactId>
            <version>0.5.1</version>
        </dependency>
```

Also we need to have Spring security in our application. For that we have to add Spring security dependency in our POM file as follows:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
<version>1.5.4.RELEASE</version>
</dependency>
```

Let's create two different URL requests. One will be secure, and the other will be insecure:

```
@RestController
@RequestMapping("/PM/")
public class SecurityDemoSecureController {

    @RequestMapping(method = RequestMethod.GET, value = "/secure/greet",
produces = "application/json")
    public String getSecureHello() {
        return "hey! secure";
    }

    @RequestMapping(method = RequestMethod.GET, value = "/unsecure/greet",
produces = "application/json")
    public String getUnSecureHello() {
        return "hey! unsecure";
    }
}
```

This is how your controller should look. Let's create configuration files of our sample application:

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

In a traditional Spring security implementation, we used to have the `AbstractSecurityWebapplicationIntializers` class. As we are using Spring Boot to start an embedded container, we don't need any `WebApplicationInitializers` interface.

There will be another configuration class about Spring security. Although we can combine this security configuration file with our main configuration mentioned earlier, for the sake of simplicity, it has been kept separately:

```
@Configuration
@EnableWebSecurity
public class SpringSecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Autowired
    privateJwtEntryPointunauthorizedHandler;

    @Autowired
    privateJwtVerificationServicejwtVerificationService;

    @Autowired
    public void configureAuthentication
    (AuthenticationManagerBuilderauthenticationManagerBuilder) throws
    Exception {
    authenticationManagerBuilder.userDetailsService
    (this.jwtVerificationService);
    }

    @Bean
    @Override
    publicAuthenticationManagerauthenticationManagerBean() throws
    Exception {
        returnsuper.authenticationManagerBean();
    }

    @Bean
    publicAccessDeniedHandlergetJwtAccessDeniedHandler() {
        JwtAccessDeniedHandler handler = new JwtAccessDeniedHandler();
        return handler;
    }

    @Bean
    publicJwtAuthenticationTokenFilterauthenticationTokenFilterBean()
    throws Exception {
    JwtAuthenticationTokenFilterauthenticationTokenFilter = new
    JwtAuthenticationTokenFilter();
    authenticationTokenFilter.setAuthenticationManager
    (authenticationManagerBean());
    returnauthenticationTokenFilter;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
```

```
            http.sessionManagement().sessionCreationPolicy
            (SessionCreationPolicy.STATELESS).and().authorizeRequests()
            .antMatchers(HttpMethod.GET,"/PM/secure/**").
            access("hasAnyRole('ROLE_Secure_Access')")
            .antMatchers(HttpMethod.POST, "/PM/secure/**").
            access("hasAnyRole('ROLE_Secure_Acces')").and()
            .exceptionHandling().accessDeniedHandler
            (getJwtAccessDeniedHandler())
            .authenticationEntryPoint(unauthorizedHandler);
```

We are adding our customize filter before the sing inbuilt filter `UsernamePasswordAuthenticationFilter`:

```
    http.addFilterBefore(authenticationTokenFilterBean(),
    UsernamePasswordAuthenticationFilter.class);


            http.csrf().disable();
            http.headers().cacheControl();
        }

        @Override
        public void configure(WebSecurity web) throws Exception {
            web.ignoring().antMatchers("/PM/unsecure/**");
        }
    }
```

As we can see, we have added a new filter `JwtAuthenticationTokenFilter` before `UsernamePasswordAuthenticationFilter`. Spring security calls this filter before the `UsernamePassword` filter. This filter will scan the header of the request and try to extract the token from the authentication header. There is an assumption that the token will be sent with the token type. That's why we break the header value on space. Normally, the token sent in the header is something like this:

```
    "bearer
    yJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Inl3U1ZJZ0VpSmtFeHJIRTN5b29jS0s5
    U0stayIsImtpZCI6Inl.....................JhKhTEE_ZTxE-
    QwkIfbHIm8LBm60DwpEYCZvyPH-kAtr7bCl-A  "
```

Here, `bearer` is the token type, and as seen in the example, token type and token value are separated by a space.

Filter class extracts the token and sends the token verification service. The following is the sample code for the filter class `doFilter` method:

```
@Autowired
AuthenticationManagerBuilder authBuilder;

//Spring auto wired Default Authentication Builder instance to this class.

@Override
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
throws IOException, ServletException {
    HttpServletRequest httpServletRequest = (HttpServletRequest)
    request;

    //Get the authorization header form request.

    String header_authorization =
    httpServletRequest.getHeader("Authorization");
    String token = (StringUtils.isBlank(header_authorization) ? null :
    header_authorization.split(" ")[1]);
    if (StringUtils.isBlank(header_authorization) && token == null) {
        logger.info("Token Not found in header .");
        return;
    }
    UserDetails principal = null;
    try {
        principal =
        authBuilder.getDefaultUserDetailsService().
        loadUserByUsername(token);
        UsernamePasswordAuthenticationToken
        userAuthenticationToken = new
        UsernamePasswordAuthenticationToken(
        principal, "", principal.getAuthorities());
        userAuthenticationToken.setDetails(new
        WebAuthenticationDetailsSource().
        buildDetails(httpServletRequest));
        SecurityContextHolder.getContext().
        setAuthentication(userAuthenticationToken);
    } catch (Exception e) {
        HttpServletResponse httpresposne = (HttpServletResponse)
        response;
        httpresposne.setContentType("application/json");
        httpresposne.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
```

Create a JSON response here to show the appropriate response to the user:

```
            ObjectMapper jsonMapper = new ObjectMapper();
            PrintWriter out = httpresposne.getWriter();
            Map<String, String> jsonResponse = new HashMap<String,
            String>();
            jsonResponse.put("msg", "Invalid Token");
            out.write(jsonMapper.writeValueAsString(jsonResponse));
            out.flush();
            out.close();
            return;
            }
        }
    chain.doFilter(request, response);
    }
```

The verification service needs three properties: `jwksBaseURL`, `jwksIssuer`, and `jwksAudience`. These three properties are needed by the identity server to validate the token. After validating the token, this service checks for the role mentioned in the token, sets the principals, and sends it back to the filter. The filter then checks whether the role found in the token is enough to give permission to access the requested object. If not, then it will send a message of invalid token back as a response:

```
@Service
public class JwtVerificationService implements UserDetailsService {
    private static final Logger logger =
Logger.getLogger(JwtVerificationService.class);
    private static final String CLASS_NAME = "JWTVerificationService";

    public static final String SCOPE_PREFIX = "ROLE_";
```

The following are the some property which should be configurable will be injecting from property file:

```
    @Value("${JwksUrl}")
    protected String jwksBaseURL;

    @Value("${JwksIssuer}")
    protected String jwksIssuer;

    @Value("${JwksAudience}")
    protected String jwksAudience;

    @SuppressWarnings("unchecked")
    @Override
    publicUserDetailsloadUserByUsername(String token) throws
    UsernameNotFoundException {
```

```
        String username = "";
        String role = "";
        JwtClaimsjwtClaims = null;
        List<GrantedAuthority> authorities = new
        ArrayList<GrantedAuthority>();
        try {
            jwtClaims = getJwtClaims(token);
            username = (String)
            jwtClaims.getClaimsMap().get("client_id");
            logger.debug(CLASS_NAME + "userName :" +
            jwtClaims.getClaimsMap().get("client_id"));
            role = (String) jwtClaims.getClaimsMap().get("scope");
            authorities.add(new SimpleGrantedAuthority(SCOPE_PREFIX +
            role));
            logger.debug(CLASS_NAME + "JWT validation succeeded! with
            Scope " + role);
        } catch (ClassCastException e) {
            logger.debug("Not able to type cast Scope in String , Tryig
            with array list for multiple Scope.");
            if (jwtClaims != null) {
                List<String>roleList = (ArrayList<String>)
                jwtClaims.getClaimsMap().get("scope");
                for (String roleStr : roleList) {
                    authorities.add(new
                    SimpleGrantedAuthority(SCOPE_PREFIX + roleStr));
                }
                logger.debug(CLASS_NAME + "JWT validation succeeded!
                with Scope " + authorities);
            }
        } catch (Exception e) {
            logger.debug("Invalid JWT !!! token = {" + token + "} found
            and exception = ", e);
        }
        return (username != null &&username.length() > 0) ?
        (UserDetails) new User(username, "", authorities) : null;
    }

    privateJwtClaimsgetJwtClaims(String token) {
        HttpsJwkshttpsJkws = new HttpsJwks(jwksBaseURL);
        HttpsJwksVerificationKeyResolverhttpsJwksKeyResolver = new
        HttpsJwksVerificationKeyResolver(httpsJkws);
        JwtConsumerjwtConsumer = new
        JwtConsumerBuilder().setRequireExpirationTime().
        setAllowedClockSkewInSeconds(3600)
                .setExpectedIssuer(jwksIssuer)
                // whom the JWT needs to have been issued by
                .setExpectedAudience(jwksAudience).
                setVerificationKeyResolver(httpsJwksKeyResolver).
```

```
                build();
        try {
            // Validate the JWT and process it to the Claims
            JwtClaimsjwtClaims = jwtConsumer.processToClaims(token);

            returnjwtClaims;
        } catch (InvalidJwtException e) {
            // Anyway here throws the exception , so no need to log the
            // error.
            // log the error if required from where this function
            // invokes
            // logger.error("Invalid JWT! " + e);
            throw new AuthenticationServiceException("Invalid Token");
        }
    }
} Credit-scoring microservice
```

Our credit-scoring service will calculate the score based on these factors:

- Payment history: 35 percent
- Current loan: 30 percent
- Length of payment history: 15 percent
- Obligation on person: 10 percent
- Recent credit activity: 10 percent

We are assuming here that the payment service is in place. This payment microservice will give information about the payment history. For simplicity, the high score is 100. The obligation service gives a number in between 0 and 10, so no need to do any operation on it. The obligation has some rules in it, defined in the following table. The assumption here is that the person is the only earner of the family:

Salary lives with	**<10000**	**>=10000 &&<20000**	**>=20000 && <40000**	**>=40000 && <60000**	**>=60000 && <80000**	**>=80000 && <100000**	**>=100000**
Alone	2	3	4.5	5	6.6	7.4	8.9
With friends	1.8	2.8	4.3	4.8	6.4	7.2	8.8
With spouse	1.4	2.6	4.1	4.6	6.1	6.9	8.6
With spouse and children	1.2	2.1	3.8	4.2	5.6	6.6	7.9
With spouse, child and parent	1	1.9	2.8	3.9	5.1	6.4	7.5

Just to keep in mind that this is a dummy table and data. Numbers and values are dependent on the data model and lifestyle in any particular country.

This will be another Spring Boot application. We are assuming here that all other microservices are using JWT for security reasons. So, we will use a service to get the token from the server, put that token in request, and make a rest call to get data from the other services.

In our POM file, we are using two new dependencies: one for converting the JSON response to a class object and one using the Apache library to make a REST request:

```
<dependency>
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient</artifactId>
            <version>4.5.2</version>
        </dependency>
        <!-- For json to Object Conversion -->
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.2.4</version>
        </dependency>
```

The controller has only one method to get the credit score for the particular user ID. It will be a very short class:

```
packagecom.practicalMicroservcies.contoller;

@RestController
@RequestMapping("/PM/credit/")
public class CreditScoringController {
    private static final Logger logger =
    Logger.getLogger(CreditScoringController.class);

    @Resource
    CreditScoringServicescreditService;

    @Resource
    ObjectMapper mapper;

    /**
     * Method is responsible for getting the account detail for given ID.
     *
     * @paramuserId
     * @return
     */
    public static final String getScore = "getScore(): ";
```

```
    @RequestMapping(method = RequestMethod.GET, value = "score/{userId}",
produces
    = "application/json", consumes = "application/json")
    publicResponseEntity<String>getScore(@PathVariable("userId") UUID
userId) {
        logger.debug(getScore + " getting information for userId " +
userId);
        return new ResponseEntity<>(creditService.getUserScroe(userId),
        HttpStatus.OK);
    }

}
```

There will be a service class that will fetch the token, talk to different services, get data from them, and finally calculate the score and send it back as a response. This particular service will require the following properties to initialize:

```
@Value("${AuthServerUrl}")
    protected String authServerUrl;
    @Value("${ClientId}")
    protected String clientId;

    @Value("${ClientSecret}")
    protected String clientSecret;

    @Value("${GrantType}")
    protected String grantType;

    @Value("${paymentHistoryUrl}")
    protected String paymentHistoryUrl;

    @Value("${obligationUrl}")
    protected String obligationUrl;

    @Value("${loanUrl}")
    protected String loanUrl;
```

The following is the definition of the fields of the service class:

- `AuthServerUrl`: This is the complete URL of the authentication server.
- `ClientId`: This is the ID that is assigned to this service by the authentication server.
- `ClientSecret`: This is the client secret provided by the authentication server. It has to be produced along with the `ClientId` while fetching the token from the authentication server.

- `GrantType`: This is the access control service over other services such as read, write, and so on. One can define its own grant type also.
- `PaymentHistoryUrl`, `obligationUrl`, and `loanUrl`: As the name suggests, they are the URLs for different services.

Now, let's have a look at the function that is supposed to get the token from the authentication server. This method puts all the values in the body section of the request and posts the request to the authentication server. In response, it is supposed to get the JWT token as a string that will be used in other services to get data:

```
private String getToken() {
        RestTemplaterestTemplate = getRestTemplate();
        MultiValueMap<String, String> map = new LinkedMultiValueMap<String,
        String>();

        map.add("client_id", clientId);
        map.add("client_secret", clientSecret);
        map.add("grant_type", grantType);

        String tokenStr = restTemplate.postForObject(authServerUrl, map,
        String.class);
        returntokenStr;
    }
```

We have a small function here: `getRestTemplate()`. In this function, we are using an HTTP factory of `bufferingClientHttpRequestFactory`. This is because if we want to log the request and response, then we have to initialize the `restTemplate` object by buffering the HTTP request client:

```
    privateRestTemplategetRestTemplate() {
        ClientHttpRequestFactorysimplerequestFactory = new
        HttpComponentsClientHttpRequestFactory(
        HttpClients.createDefault());
        ClientHttpRequestFactoryrequestFactory = new
        BufferingClientHttpRequestFactory(simplerequestFactory);
        RestTemplaterestTemplate = new RestTemplate(requestFactory);

        returnrestTemplate;
    }
```

In the next method, we will call three different URLs and send back the response received by hitting the URL:

```
/**
     * Method is responsible for getting the Payment History
     *
```

```
        * @paramuserId
        * @return
        */
   private List<String>getPaymentHistory(UUID userId) {
           String token = getToken();
           if (token == null)
               return new ArrayList<String>();
           RestTemplaterestTemplate = getRestTemplate();

           HttpHeaders headers = new HttpHeaders();
           headers.add("Authorization", "bearer " + token);
           headers.add("Content-type",
           ContentType.APPLICATION_JSON.getMimeType());
           headers.add("Accept", MediaType.APPLICATION_JSON_VALUE);
           HttpEntity<String> request = new HttpEntity<>(headers);

           ResponseEntity<String> result =
           restTemplate.exchange(paymentHistoryUrl +
           userId, HttpMethod.GET, request, String.class);
           ArrayList<String>responseObject = new
           Gson().fromJson(result.getBody(),
           ArrayList.class);
           returnresponseObject;
       }

       /**
        * Method is responsible for getting the Obligation scrore
        *
        * @paramuserId
        * @return
        */
       private Integer getObligation(UUID userId) {
           String token = getToken();
           if (token == null)
               return 0;
           RestTemplaterestTemplate = getRestTemplate();

           HttpHeaders headers = new HttpHeaders();
           headers.add("Authorization", "bearer " + token);
           headers.add("Content-type",
           ContentType.APPLICATION_JSON.getMimeType());
           headers.add("Accept", MediaType.APPLICATION_JSON_VALUE);
           HttpEntity<String> request = new HttpEntity<>(headers);

           ResponseEntity<String> result =
           restTemplate.exchange(obligationUrl +
           userId, HttpMethod.GET, request, String.class);
           returnresult.getBody() == null || result.getBody().length() ==
```

```
        0 ? 0 :Integer.parseInt(result.getBody());
    }

    /**
     * Method is responsible for getting the Obligation loan score
     *
     * @paramuserId
     * @return
     */


    private Integer getLoansList(UUID userId) {
        String token = getToken();
        if (token == null)
            return 0;
        RestTemplaterestTemplate = getRestTemplate();

        HttpHeaders headers = new HttpHeaders();
        headers.add("Authorization", "bearer " + token);
        headers.add("Content-type",
        ContentType.APPLICATION_JSON.getMimeType());
        headers.add("Accept", MediaType.APPLICATION_JSON_VALUE);
        HttpEntity<String> request = new HttpEntity<>(headers);

        ResponseEntity<String> result =
        restTemplate.exchange(obligationUrl +
        userId, HttpMethod.GET, request, String.class);
        returnresult.getBody() == null || result.getBody().length() ==
        0 ? 0 :Integer.parseInt(result.getBody());
    }
```

All these methods are there to get the data from different web services. Also, they are all secure calls as they are with JWT and encryption. Now, with all the data, a simple mathematics credit score can be generated. The obligation score can be used as it is. With payment history, one can identify the failure or delayed payment. Let's say it has 56 failures in the payment history, then we will have another table as obligation and get the number from there. In this example, 56 failures in the history is from more than 500 payments. Almost 10 percent of the payment is failure. We can calculate the percent for payment point with this 10 percent and add some predefined factors in it. Let's assume that the total is 45 percent. 100 - 45 = 55 percent points (out of the total weightage of payment) should be given to the user for credit. As payment history has a weightage of 35 percent, we can calculate 55 percent of 35 percent, which come to ~19.25. By adding other scores to it, we will get the total credit score of the user.

The formula mentioned earlier is a dummy and explained as an example. In an ideal case, this formula includes much more statistics and machine-learning algorithms that improve with more data. This is the example of a simple credit-scoring application. We can make it more effective by making an effective data store.

Summary

In this chapter, we discussed common security challenges among microservices. Some of the security hygiene practices should be followed in microservice architecture. We understand the OpenID flow and OAuth 2.0. We have also introduced the JWT structure and its usage in the sample application. In the next chapter, we will discuss an effective data model and also discuss our application data model.

5
Creating an Effective Data Model

The software industry is rapidly changing. Rules, dimensions, and laws are changing daily. To gain more knowledge about the industry, you need to reiterate fast, build, get feedback, improve, and so on. Moving at this pace in a monolithic environment will make things more complicated. Microservices makes this path smooth. Going by the microservice architecture and breaking big problems into smaller ones can lead us to distributed components. Working in a distribution or scalability area always challenges the traditional ways of doing things. The ripple effect of distributed architecture can be seen on the database also. The traditional data model can't be applied to distributed architecture. Another challenge is dependencies. When problems become smaller and distributed, it is challenging to maintain the dependencies among different components. These dependencies can be in terms of the data model, coding library, and so on. So, what is the best database model for microservice architecture? There is no definitive answer that is well fitted to all situations, but we will try to discuss the various factors that affect the decision of data modeling in the upcoming sections of this chapter.

In this chapter, we will cover the following topics:

- Microservice data model structure
- Comparison with a traditional data model
- Intermixing data technologies when needed
- A walk-through sample application data model

Data and modeling

Data is everywhere now. Every business needs data, runs on it, and decides the current and future strategy for business by analyzing historical data. Data is very important, but not all data/information is necessary or relevant to a particular business. So, not all data should be used or stored by the system. One has to identify the data that is important to business, store it, and then provide knowledge, using it for future reference. Once we have identified the data that is important, how we should store it? Should we do it in the raw format or define a structure to store that data? To make sense of and gain knowledge data, it has to be stored in a structured format. The process of identifying and defining this structure is data modeling.

According to Wikipedia, a data model is:

> *An abstract model that organizes elements of data and standardizes how they relate to one another and to properties of the real-world entities.*

Data modeling is a visual representation of organizational data semantics. It includes entities involved in businesses, such as customers, products, orders, and so on. It also includes the relations between them. Entities are the main factors to store in the database, along with their attributes. For example, for customer attributes, there are customer name, address, phone number, and so on. Another thing to consider is cardinality, which represents the degree of relationship. For instance, a customer can order multiple products and also place multiple orders. So, the cardinality between the customer and their orders is one-to-many. Many-to-one and many-to-many are other forms of cardinality.

In other words, we can rephrase the sentence as, *a data model is a relatively simple representation of complex real-world data*. At every level, the user requires different views of data. So, a good data model should be capable of fulfilling the needs of all different users with different permissions.

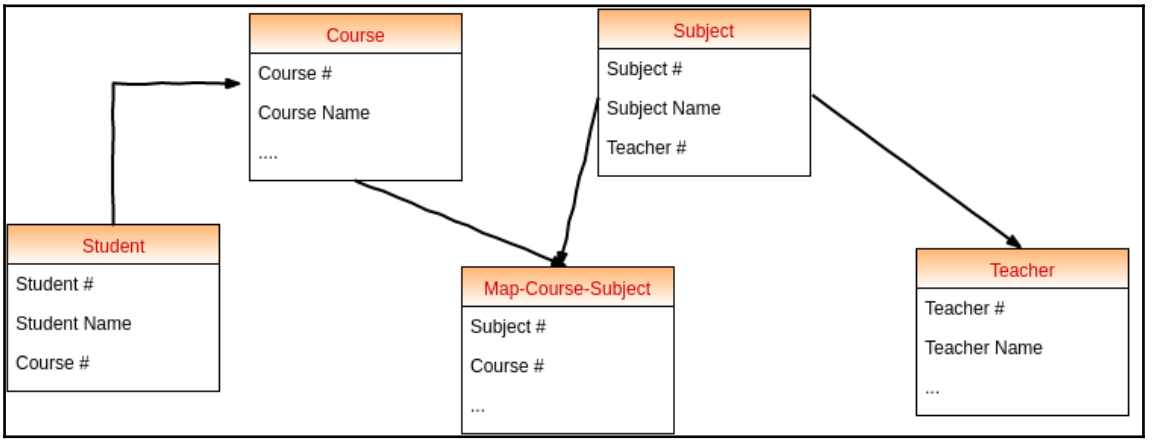
Comparison with a traditional data model

Over a period of time, software engineering has evolved. It has taken the database modeling/design on the same path. So, database modeling has also evolved on this path. Starting from monolithic to service-based architecture, there are different milestones that database modeling has achieved. In this section, we will talk about database modeling done in a monolithic application and in SOA. Then, we will discuss data modeling done in microservices. Keep in mind that the modeling techniques discussed here are not the only ones used. There could be other techniques as well, and we do not prefer one technique to another. The modeling techniques discussed here are to give you an idea about how, most of the time, things are done in the most commonly adopted way. This model was widely used around the world in its era.

Data model in monolithic architecture

In monolithic applications, all modules are bundled in one big fat application, such as WAR/JAR/EAR. If one module has to use another module's functionality, then it will make a simple call to that module. This is how module dependencies are created. Modules share data among themselves. This is the way dependencies are created among various microservices. They might be using the same one-instance database. This eventually creates tight coupling between modules by sharing the common data model. That's why a database model can be seen as one large database behind an application. In a traditional data model, the journey starts from an ER diagram and ends with a relational model. The relational model basically includes tables and relations between tables. Attributes are defined as fields or columns of these tables. One row of a table in the relational database is called the **tuple**. Every tuple must have a unique key that define its data uniquely. These keys are used to join data from two or more relations. The foreign key concept is also included in the relational model, which easily makes relations or connections.

As shown in the following basic example, let's take a situation where an organization wants to offer courses to students and wants to record information. The first step will be to identify the important entities in the system, such as courses, students, subjects, and teachers. Students can enroll for a course, and a course can have one or more subjects. Teachers can teach one subject (for now). So, we can see our identified entities as relations. The course table will store data related to courses, and student relations will store the data of students who enroll for the course. The mapping of each course, with subject is stored in a map-course-subject table . This is a very simple relational model example:



You need to understand how to see the data here. Data is seen as an entity and its relation with other entities. Relationships can be of two types: **IS-A** and **HAS-A**. Just as courses has (HAS-A) subjects. A HAS-A relation means one entity (course) has to have another entity (subject) inside it to complete itself. On the other hand, an IS-A relationship happens to be in between the same type of entity. For instance, a Volvo IS-A is type of vehicle. So, between vehicle and Volvo an IS-A relationship exists.

This whole approach mentioned in the preceding scenario is very good. However, since there are so many relations between entities, data has to stay in one place to fetch any meaningful data, so that join queries between relations can be executed and constraints such as foreign key can also be applied to different tables. Using this approach, all the data of an application is tightly coupled. Changing a data model needs a release of the whole system again.

Data model in SOA

The previously mentioned data model is mostly used in traditional, single fat applications architecture. In the next step, where we come to the SOA era, where we get a flavor of services-based architecture, there is a new data model introduced, named the **Canonical Data Model** (**CDM**). It is used in the SOA concept, which advocates the reuse of data structures, attributes, and data types during messaging between various systems and components. The profit of this concept is that it simplifies the definition of the business data, but it could easily become unmanageable. Whatever kind of technology one is using (an ESB, a BPM platform, or just an assembly of services of some kind), standardize the data models of the business objects you exchange. Due to OOPs mindset, in very common scenarios, you end up having some kind of entity, such as customer, order, product, and so on, with a set of IDs, attributes, and associations everyone can agree on. It seems like a very convenient thing to do. Then, anyone who happens to have a model that differs from the canonical one will have to implement a conversion to and from it.

There are benefits to using a canonical data model. For example, CDM promotes the use of a common definition of the objects across the business unit in an organization. This really helps with better communication and understanding, and reduces the chance of ambiguity. Also, integration is very easy and cost effective in this model, in terms of time and the developer's effort.

Data model in microservice architecture

The notion of a CDM is rejected by the microservice integration pattern because microservices do not require them. CDMs encourage users to share data through a common channel. If we go by this idea, then a user will bind themselves to using the same technology in all microservices, which is eventually contradictory to the polyglot nature of microservices. If you share the model, you are coupling microservices and will lose one of the greatest advantages in which each team can develop its microservice without restrictions and the need of knowing how others' microservices evolve. Major things to keep in mind in microservices data modeling are:

- **Loose coupling**: Any microservices should be able to be modified without affecting any other microservices
- **Problem locality**: Identified problems should be grouped together

One thing leads to another from both of them. For loose coupling, we need a clean contract about the model of data.

The microservices approach to data modeling is quite distributed and independent among themselves. As a principle, in microservices architecture, all the services are modeled as independent and isolated services. This principle impacts the decision of database modeling as well. If one has analyzed the problem statement or system in microservices perfectly, then any microservice can be developed and deployed in parallel and independently. To achieve this state, each microservice data model should be independent of other microservices data models. Any change in one data model should not affect another service's data model.

Restricted tables for each microservice

This situation leads us to the idea of each service having its own private data. Only that service should be allowed to fetch, update, or modify that data. This can be achieved in different ways. Each microservice can have a set of tables assigned to them. Data in these particular tables can be modified by only a particular service. To implement this idea, the service can have a user defined in the database who is allowed to alter certain tables. This way, we can ensure that each service does not disturb the data in tables that are related to other microservices.

The idea of having tables in one database and having authorized access to those tables seems good, but again, it will bind us to one kind of database, such as Oracle or MySQL. If one team wants to try another kind of database for performance, they wouldn't be able to use it in this scenario.

Database per microservice

The next idea is each microservice having its own database. This gives another free hand to the developer to choose the database type as per the need of the microservice. The developer need not stick to the only database technology used by other microservices. This seems something like distributed CDM. This seems to be a convincing thing to do. However, it has its own challenges. Tight coupling is not introduced only based on shared data, rather it also depends on data flow among different services. To keep the system in a consistent state, one must execute all the commands to fulfill the request or roll back to the original state if any step fails. This is comparatively easy in a traditional data model that has a lock on tables, with commit and rollback in one database. In a distributed environment, each component works in isolation, rolling back from any step in the middle to the initial state.

For instance, normally, business transactions (perhaps financial) survive on multiple services, and multiple services have to update data. Then, it becomes very challenging to handle failovers efficiently. One solution to it is, rather than using distributed transactions, you typically must use an eventually consistent, event-driven approach to maintain database consistency. There is one proposed solution for this: using the Saga pattern. This pattern was used by Hector Garcia-Molina and Kenneth Salem in 1987. They designed this solution keeping in mind a very loosely coupled environment. They tried to produce the single transaction effect in a distributed system. Saga seems to be a very compelling pattern to use. Although it doesn't make the transaction the same as the initial state, it makes the system consistent and brings it very near to the initial state. Other than Saga, we can think of a two-phase commit in a distributed environment, but it doesn't seem to be an ideal solution in the case of scalability.

The Saga pattern

The Saga pattern gives a transaction management solution in a distributed environment, where we have different subsystems (microservices) communicating with each other. There are many long-lived transactions in a distributed environment in the real world. Any transaction makes sense in a database if it works completely or fails completely. In the first scenario, our system comes into the consistent state automatically. The main challenge is to bring our system into the consistent state after a transaction fails at any subcomponent.

In Saga, every workflow of a transaction is grouped in a composite task that we call routing slips, which are handed along the activity chain. When an activity completes, it adds a record of the completion to the routing slip, along with information on where its compensating operation can be reached (for example, via a queue). Alternatively, it may register a callback to a compensating transaction in a message routing slip. Finally, it passes the updated message down the activity chain. When an activity fails, it cleans up locally and then sends the routing slip backwards to the last completed activity's compensation address to unwind the transaction outcome. The previous step does the same thing, calling its predecessor compensating transaction, and so on until all the already executed transactions are compensated. Due to their highly fault-tolerant, distributed nature, Sagas are very well-suited to replace traditional transactions when transactions across microservice boundaries are required in a microservice architecture.

The very first component in a Saga transaction pattern is called an initiator, which creates the context of transaction and the reason for the interaction. It then asks one or more other services (participators) to perform some businesses activities. The participators can register for coordination and exchange messages and requests until they reach some agreement, or they are ready to complete the interaction. This is when the coordinator requests all the participants to finalize the agreement (prepare) and commit. If there are any problems during the completion of any activities, you have to perform a counteraction, called compensation, just as you have rollback in regular ACID transactions. In coding layman's language, I should like to rephrase it as: every step or every microservice performs its task, marks its task as done, registers its callback method in the case of failure in any upcoming activity (microservice), and passes its route slip to the next microservice. If anything fails, then the routing slip is passed back to the last completed task by calling its compensation method.

Let's understand it using an example. A user wants to book a table in a restaurant with a specific menu for dinner, and they also need to book a cab to reach the restaurant on time. Here are the steps the user will follow:

- Check whether a table is available in the restaurant. If not, the transaction will not propagate from here.
- Look for the desired menu item. If it is not there, there is no point reserving a table. So, the last activity has to roll back.
- If the table and menu are booked, then book a cab for a particular time. If no cab is available during that time, the user has to cancel the reservation also. So, the last two activities can't be committed; they have to roll back.

The assumption here is that the user is using the same website to make all these reservations in one go.

The Saga pattern is an architectural pattern that focuses on integrity, reliability, and quality, and it pertains to the communication patterns between services. When it comes to designing the implementation of the pattern, everyone has a different version. You can implement the concerns and roles defined in the pattern in a method of your choice. There could be one component, maybe the initiator handling the controls of the flow, like an orchestration pattern, or it could be a distributed pattern.

It is important that each service's persistent data is private, but if we go by **database per service** strategies, it might be difficult to implement some queries if they have database joins across the data owned by multiple services. Joining data inside a service is possible. In other situations, you will need to use **Command Query Responsibility Segregation** (**CQRS**) and maintain denormalized views.

Intermixing data technologies when needed

Polyglot persistence is a term used in the context of microservices. This term was coined by Scott Leberknigh and Martin Fowler. An enterprise can have different types of data model and access frequency. Some data is required very frequently, and we need not bother much about their accuracy. On the other hand, some data needs to be persisted in a very certain manner. There are many more such different type of conditions, which may occur with different data. Those situations are the reasons enterprises have different technologies to store different types of data. For instance, for sessions, they use Redis; for recommendation engines, they might use Neo4j or any graph database; for finance transaction management, they use some kind of RDBMS such as MySQL or Oracle, and so on. This situation is referred to as polygot persistence.

Any e-commerce website normally has a product catalog to show to the user. When a user selects any one product online, as a business person, you want to show them another product similar to that one or a product that can be used with the selected product. The customer sees some messages such as "XX product is normally purchased together with YY product." It is like having a virtual stubborn sales guy who keeps showing you the things that can be used with your selected product until you make the payment of your selected product. If this is the requirement and tech has to use SQL-based databases, then it will be hard (not impossible) to implement it efficiently. On the other hand, if they choose any graph database, then making links among the products frequently brought together is very easy and can keep on improving. So, the user management part can be an SQL-based database, and the recommendation engine can be another type of database.

In brief, there are, however, some applications that benefit from a polyglot persistence architecture that uses a mixture of database types. A downside of not sharing databases is that maintaining data consistency and implementing queries is more challenging.

Migrating a data model from monolithic to microservices

A typical enterprise application can have three parts: frontend with some HTML, CSS, angular stuff; a server part that handles requests and does some business logic; and a database part that has tables for all the contexts we have in our application. If you need to add one new thing to a database or at server side, even if it is in a tiny and isolated part of the application, you have to deploy the whole application. It tends to cause downtime, which is not very acceptable in today's era.

In these kind of scenarios, microservice comes up with a solution. However, migrating monolith to microservices is not easy and includes lots of decisions and considerations. Here, we are mainly talking about migrating from a database model of monolithic architecture to microservices-based architecture. One can go with a big step and convert the whole monolith application into microservices. Another option is go step by step and find a piece of a data model that can be used as a separate data model without affecting much of the whole application's data model.

The main concern of migrating from a monolith database is confusion while extracting your microservice databases. There is no foolproof plan for this. There are patterns that help us understand what approach we can use to create our microservices data model. While breaking monolith to microservices, you know better which data mostly updates together, or you can take the help of domain experts too. Let's understand Domain-Driven Design better.

Domain-Driven Design

If we go by database per microservice, then how do we identify which data bucket has which kind of data? DDD helps us in this. However, when we use the words *data bucket* here, this term should not be treated as a physical database such as MySQL or Oracle. Here, we are referring to a database as a collection of similar kinds of data. Data modeling and how data should be stored in a table are two different problems. Data modeling is the concept of thinking in terms of what data is necessary to deal with a particular problem domain. The priority should be a focus on solving the business problem first, then the persistent technology. DDD is more about focusing on the business domain first.

To explain this better, we will take the example of Amazon. Amazon has a large number of products and displays their relative information, user data, order details, shipping information, current location of order, support requests, and so on. This data is in the region of hundreds of thousands of petabytes and is continually increasing.

The Amazon team is facing the challenge of storing this data, geting fast access to it on a daily basis, to load product details quickly, and so on. If you think it over, it is not the business problem they are solving. The business problem they are solving is letting users anywhere in the world check out the product sold by buyers anywhere in the world, connecting them, showing users as many options as they want, and making sure they get the product they ordered while sitting at home.

Hypothetically, if Amazon starts to solve this problem using DDD, then they must figure out the different contexts in their system, such as user management, product/categories management, order management, reporting management, payment and refund management, shipping management, sellers management, and so on. Now we have broken the bigger problem into small bits. The next step will make us think again about these problems, or we can break them down into smaller bits and understand how data will flow from one component to another. For example, one user could have two or three addresses, so we need address management. Customers can save their cards to pay quickly, so there could be card management also in this subdomain. Then, the next step is to decide which data is relevant to the store and which data could be relevant to some other context. In a nutshell, user-related data should be here. So, this is the domain and its bounded context. So, the next step would be to store user management data in an SQL or non-SQL or graph database. This is the next challenge.

DDD is a good candidate in terms of solving business problems in a microservice system. The DDD concept has been around since before the microservice was born, but it matches perfectly with microservice. The microservice data model works best with the concept of DDD. As mentioned before, DDD focuses on the business problem first. For that, the first step would be identifying the different contexts in your problem domain. Domain experts can help here, with their experience and knowledge. Once the contexts are identified, the next step will be to define the boundaries of these contexts.

Bounded context is the most important factor in DDD. Bounded context is an effective way of designing microservice boundaries, and DDD techniques help in breaking up a large system along the seams of those bounded contexts. If we use DDD and bounded context approaches, it decreases the chances of two microservices needing to share a model and the corresponding data space, or ending up having tight coupling.

Avoiding data sharing improves our ability to treat each microservice as an independent deployable unit. It is how we can increase our speed while still maintaining safety within the overall system. DDD doesn't advocate sharing data among the models. However, if we still have to do it, then there is the concept of the **anti-courption layer** (**ACL**), which is able to transform data from one format to another. There are other better ways of doing it, such as event sourcing, which we will discuss later.

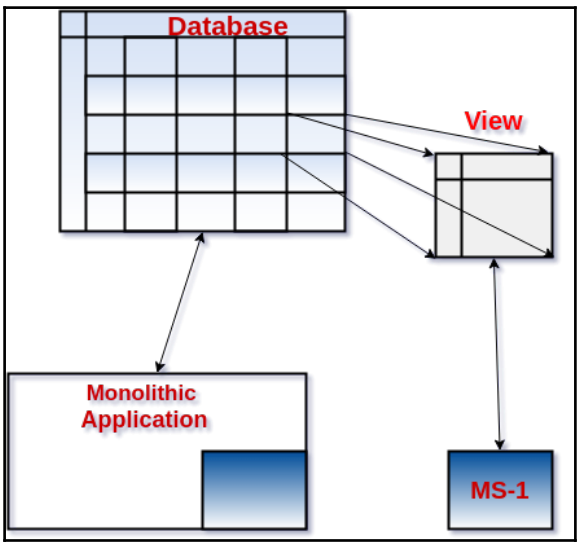
Methods of data model migration

As we mentioned before, data model migration can be broken into smaller steps too. There are ways that help us create a separate data model from a monolithic application:

- View
- Clone table using trigger
- Event sourcing

Views

A database view is a subset of the database sorted and displayed in a particular way. For each view, we can control the row (by `where` condition) or column (in `select` statement) to be shown. View mainly depends on database technologies. One doesn't have to extract the table from the main database. The developer can start it by creating a view and pointing code to the view. The benefits are it is the easiest to start and many databases support this concept. The only concern here is performance during updates. We can increase the performance using a materialized view. Again, it depends on the database support. As shown in the following screenshot, you can break one part of code as microservice and create a view on the database. Microservices use this view as a source to get and update data. Run this setup for a few days to verify its domain breakup knowledge:



Clone table using trigger

One can use a new database with microservices and start filling data in the new database using trigger. It largely depends on database support. It will give strong consistency, but a tool such as DBlink is required and one DB is reached by another DB. There is a strong chance of you ending up using the same database technology and you will not be able to profit from polyglot persistence.

Event sourcing

A powerful data-modeling methodology that can help us avoid data sharing in microservices, even in very complicated cases, is event sourcing. This is one of the best ways to deal with data modeling in a microservice. An event is an action that happened in the past, and it's not a snapshot of a state. An action itself is equipped with all the information that's necessary to replay it. Data in a database is actually a state gained after performing a stream of events. So, the assumption here is that the same database state can be achieved if the same set of events are processed in the database in the same order. Examples of event sourcing are version control systems that store the current state as differences. The current state is your latest source code, and events are your commits.

Another very interesting fact is that events that have taken place are immutable. An event describes something that happened in the past and thus cannot be undone. For auditing and reporting, it becomes a perfect use case. It brings eventual consistency to the system. So, if you need a strong constant system, event sourcing may be not for you. In simple language, eventual consistency means the system does not ensure being in a consistent state every time. It might give outdated information for some time, but eventually, over a period of time, it will become consistent with its information. That period of time could be 10 seconds or 10 hours; it purely depends on use case and implementation.

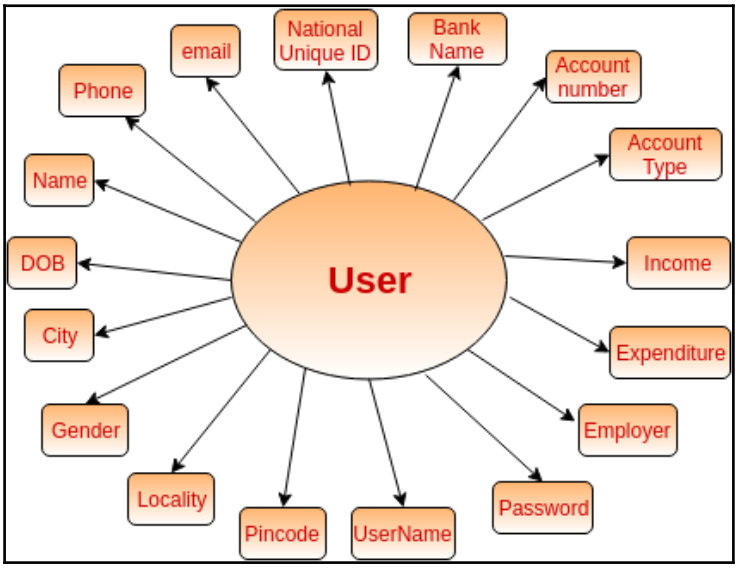
If you are moving from a monolithic application to a microservice-based application and have decided to use event sourcing, then there is a very nice tool called Debezium. It streams out the changes happening in your database as events. The microservice component can read this event and utilize it. For more information on Debezium, you can go to `https://github.com/debezium`.

The success of microservices architecture success lies in the secret of how good one can decompose bigger problems into smaller ones. There is a strong chance we may miss something while breaking down to smaller things from a monolithic to a microservice-based application. So, the golden rule is never do destructive documents.

Sample application data model

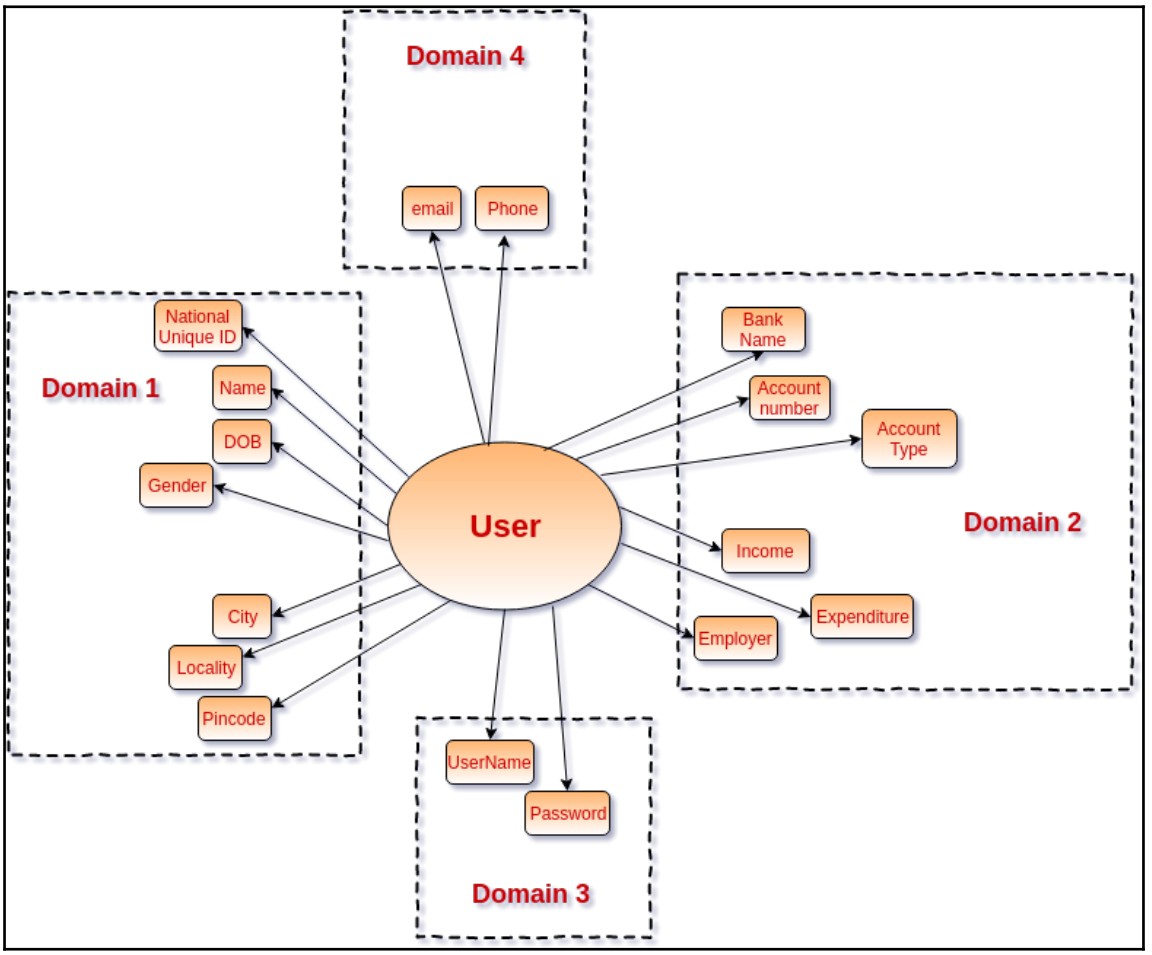
Our sample application is a risk engine, which mostly have some rules. Based on the data we have, we run a rule on it and try to find out a score. On the basis of that, we will find out how much risk is involved in lending money to a particular user. We will try to conceptualize what a risk engine component is, define a database, and use some DDD on those to come up with a microservices-based data model.

A risk engine has to have user details. Users have many details, as shown in the following diagram. Users can have much more data, but we are showing this as a sample application to make you understand how we should think in terms of domain:



If we look at the preceding data model example, we can see that all the information is related to a user. The question is, does all the information come in one domain? Or can we figure out different domains in these data attributes?

We can see four domains or segments in that information. We divided these domains or segments based on data which seems to be related and normally changed together. The following diagram shows the domains we can identify:



Dividing the domains fairly depends on knowledge and expertise. Let's explore these domains and understand why we chose them.

Domain -1 (Details)

It basically deals with user details. The minimum information required to register a user. It needs name, gender, date of birth, unique ID, and address. We put these in the same domain because the probability of changing this data at the same time is high. Updating this data should not affect the other data models. If someone updates the user's address, it should not affect the bank account number, income, and so on.

Domain 2 (Financial)

Domain 2 mainly consists of financial information. It could be bank details that the user is using, and also the income of the user. Both types of information are somewhat related. For example, if a user is a salaried employee and they change their job, there is a high probability of changing the income and also the bank account. As some companies open salary bank accounts in their collaborated banks, that makes the probability of having a new bank account with a change in job very high.

A new company may ask a user to create a new bank account in their collaborated banks. That's why we have this information together. One interesting thing here is a change in employer can change the user address also. So, should we add the address here? It mainly depends on past experience and the dataset we have for the target user. That's where domain experts come into the picture. One could think in a way that people with a lo of experience or a wide skill set must have purchased a house and have a family. These kind of users do not change their address very frequently. So, you have to identity the dataset and try to find a pattern in the data that changes together.

For our application's risk engine, we are assuming we will be giving a high bucket loan. A high bucket loan is a risk that can be taken only if users have a high income and also the chances are that the users have purchased houses already. So, there are less chances that the address of users change frequently. So, keeping this probability in mind, we are not keeping addresses with income. There should be the right balance between domain and data set pattern knowledge to make this work.

Domain 3 (Authentication)

Domain 3 is a simple domain for authentication. The application should have a unique way of authorizing and authenticating the user. That's why credential information is put together as a different domain.

Domain 4 (Communication)

Domain 4 is more to deal with channels from where the application communicates with users. In the preceding data model, we can see the phone number and email are used to communicate with users and both of them fit in the communication domain. If we have a mobile app or want to store the device IDs of those, we can do it here.

Now we can think of moving them to storage. In our use case, all can go to a relational database. We can have different databases for each domain and can have tables to store data. Domains will be connected by user IDs. Whenever a new user registers, their information will be published with a create event and user ID. All other services will be subscribed to events. After listening to the event, they can make entries in their database also. Generations of user IDs can be the responsibility of the first component. It will be used by all other components.

This is one way we can start modeling of our application. You may have gotten an idea about how to start thinking in terms of domains and datasets, rather than a relational model or storage unit first.

Summary

In this chapter, we touched on data modeling done in a different era. We discussed some of the common problems in traditional data modeling if you were working in a microservices environment. You were introduced to a pattern for long-running transactions, which was the Saga pattern. Also, we looked at DDD. Further, we saw how to model a sample application in a microservice environment. In the next chapter, we will explain how to do testing in microservice-based architecture.

6
Testing Microservices

An increase in the number of microservices in a platform will increase the DevOps work exponentially. In this situation, automation will take the lead, bringing challenges in automation testing. Unlike in monolithic applications, it is not straightforward to test microservices. As microservices can receive API calls from many other microservices and make calls to other microservices, testing a microservice in isolation is tricky with integration testing. It requires a considerable amount of effort to plan microservice testing. In this chapter, we will go through some techniques that we can use to test a microservice.

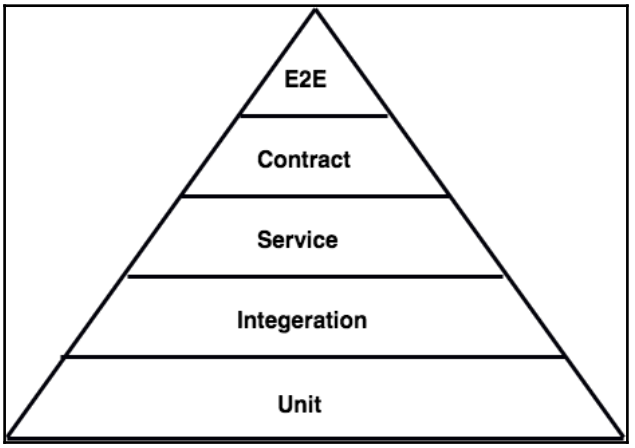
We will cover the following topics in this chapter:

- The purpose of testing in the microservices world
- Unit tests
- Integration tests
- Component (service) tests
- Contract tests
- End-to-end tests

The purpose of testing in the microservices world

Microservices are different from monolithic applications in many ways, so the approach to testing microservices can't be the same as that of testing monolithic applications. As releases are more frequent in microservices, speed is an important factor. To keep up the speed, testing of microservices should also be fast, but quality is also a major factor in microservice testing. Testing one microservice in isolation, though tricky, is a good practice -unit tests can help us here. However, this is not the case all the time. A microservice is used to communicate with a third-party sometimes, and also other microservices; so unit testing is not sufficient here. Any change in service A can impact the communication between service A and service B. So, the developer/QA has to ensure that nothing is broken during this development process. Many teams are working on different components, but they impact communication between other microservices, so it is essential to have contract testing in place. The QA is supposed to perform end-to-end testing on the platform; some of these tests can be automated, while others are manual.

From this discussion, it is clear that no one method is sufficient to ensure the compatibility, integrity, and quality of newly developed microservices. To ensure all this, we will combine some of the best-suited testing strategies to accomplish our objective. The layers shown in the following diagram are not fixed or suitable for all organizations, but this can be treated as a model for testing paradigms in microservices:



Every organization makes changes in this layer structure according to their team and requirement. Testing starts with unit testing and ends on **end-to-end** (**E2E**) testing. We will go through each layer in the upcoming sections.

Unit testing

Unit testing has been around for quite some time now. It works on the principle of validating an assumption in code. So, even if the code changes after some time, the assumption should remain valid. As it used to test a particular unit of code with a particular assumption, it is referred to as a unit test case. Let's suppose that you are writing a Java class and there are 10 methods in that class, each of them serving 10 different purposes. Then, we can say that you have 10 units of code to be tested. To go with the example, one of the methods is taking ID as input and returning back a Boolean about that ID: is it enabled or disabled in the system. Following is the sample code for a method which checks whether the given ID or, say, user ID is enabled or disabled. :

```
public boolean isEnable(String Id){
   return someRepo.findOne(Id).getStatus().equals("ACTIVE");
}
```

The expected output from this method is either `true` or `false`. What happens if the ID does not exist in the database? Should it raise an exception or give a value? Again, the developer has to look back at the assumption. If the assumption says that it should return `false`, then the developer has to make a change to the code. This is how unit test cases evolve.

Let's look at a very common example of a unit test case, a calculator. A calculator provides arithmetic operations on two integers:

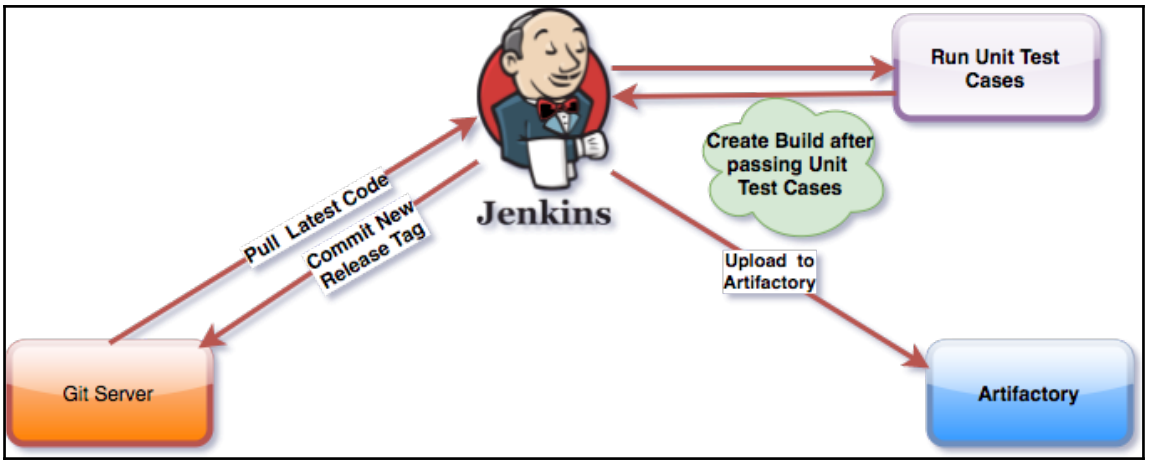
```
public class Calculator {
   public static int addition(int number1, int number2) {
      return number1 + number2;
   }
   public static int subtraction(int number1, int number2) {
      return number1 - number2;
   }
   public static int multiply(int number1, int number2) {
      return number1 * number2;
   }
   public static int divideInteger(int number1, int number2) {
      if (number2 == 0) {
         throw new IllegalArgumentException("Cannot divide by 0!");
      }
```

```
        return number1 / number2;
    }
}
```

Here are some sample test cases for the preceding code:

```
public class ExampleCalcTest {
    @Test
    public void shouldPass_forAdditionOfNumber() {
        assertEquals("error..",  8, Calculator.addition(1, 7));
        assertEquals("error..", 23, Calculator.addition(17, 6));
        assertEquals("error..",  17, Calculator.addition(8,9,));
    }
    @Test
    public void shouldFail_forAdditionOfNumber() {
        // assertNotEquals(String message, long expected, long actual)
        assertNotEquals("error..", 0, Calculator.addition(13, 2));
    }
    @Test
    public void shouldPass_forSubstractionOfNumber() {
        assertEquals("error..",  5, Calculator.subtraction(12, 7));
        assertEquals("error..", 3, Calculator.subtraction(4, 1));
          }
    @Test
    public void shouldFail_forSubstractionOfNumber() {
        assertNotEquals("error..", 0, Calculator.subtraction(2, 1));
    }
}
```

The TDD approach supports unit test cases. These test the cases only to validate the logic inside a particular unit of code in the microservice. Depending upon the classes and methods inside the microservice, the number of these test cases can be high or low. Tests are automated test cases. If we take an example of a Java + Maven project, then it will run at build time. The Jenkins build tool will fetch the latest code from source version control (for example, Git). In the next step, on the latest pulled code, unit test cases are run, after the successful completion of those test cases, builds are made and uploaded to the artifact repository and a new tag is committed in the Git code. If the unit test cases fail, then the build is marked as failed and its feedback is given to us. The following image depicts the preceding steps:



Unit test cases cover a small part of code. Depending on how much TDD has been focused on, the number of test cases increases. They target a very small part of code and its assumption, so if any problems occur in the unit test cases running for an application, then it is easy to identify the exact cause of the problem because whichever test case failed is responsible for testing a small piece of code; identifying issue in a small piece of code is faster.

The required dependencies in the particular code which has to be tested should be mocked in unit test cases of that code. There should be the appropriate output expected from these mocked dependencies with respect to input. There could be a rule in mocked dependencies such as should respond *X output for Y input*, which makes test case writers focus on small parts of code for which the test case is being written. Until now, we have tested small chunks of code in isolation, assuming that all the other units of code are working properly. Going forward, we need to test how these different code units work together. For this, we need to perform integration testing, which we will cover in the next section.

Integration testing

Unit test cases perform test checks and verify the code, but they only verify the inside logic of that particular part of the code, not the dependent module. For example, if service layer A method depends on service layer B method, and we are writing test cases for service A, we will mock service B. In this case, service layer A passes all its test cases and the same happens with service layer B. So, test cases for both service layers run perfectly fine individually, but not with each other. So, the verification of service layers is done on an individual level, but communicating with other service methods is also tested in the case of microservices.

This is where we introduce integration testing. It ensures that the all the different layer components work perfectly together and provide the expected results. For example, the data layer is getting data from the database, and the service layer is hitting the database layer and converting data into the desired format to send back to the user. In an ideal scenario, a service layer has its own unit test cases, as a service layer has dependencies on the data layer, so the data layer is mocked in service layer test cases. As service layer test cases only focus on service layer code, there could be a test case where the service layer sends some input to the data layer which is not acceptable for the data layer in reality. The reason could be anything and completely independent of service layer logic, say perhaps due to a validity fail in the data layer. But as we have mocked data layer in test cases, it will accept and respond with a predefined response. This results in successful test completion of the service layer. However, in the test case, it will fail, as the value sent by the service layer is not acceptable for the data layer method.

In the same way, there could also be external service calls in the system, as the microservice architecture includes many services running together. So you might ask whether testing with external services (external to your service code) should be included in testing. I would say no; any external service should be mocked at this stage. As a developer, you are not responsible for other people's code. Although some of you may not agree with me on not including external service in integration testing, I feel that it is a part of end-to-end testing, which we will cover in a later section.

In integration tests, the tester or developer is responsible for the entire subsystem test case - not just a small portion of the code. It targets the whole module. So the impact of these types of test cases is more than that of unit test cases, as they try to analyze the behavior of the entire subsystem. They interact with peer methods or layers in a service to make sure that they communicate in the way they are supposed to.

Component (service) testing

In the microservice context, we can say that any service itself can be considered as a component. Here, we can say that testing the whole microservice is component testing. As per its definition, *a component is any well-encapsulated, coherent, and independently replaceable part of a larger system*.

This testing has to be done on an already deployed microservice, which is supposed to be a good representation of the production environment. So, this service or component definitely can't hold a specific logic for testing.

As it includes real network calls or database calls, a database can point to testing the database through configuration injection. It actually increases the work for the tester to start and stop the test databases and the various other stubs and configurations. As it runs in a close-to-real environment, it will face the reality issue too, which we think is not necessary to handle at this stage; for example, it can have a database interaction, which can be time consuming for a testing plan, or a network call, which is also time consuming and sometimes even faces the packet loss issue.

One option to overcome this is by using a test clone of the original external service. Now, the question is, whether these test clones should be injected by the testing environment or they should come into the RAM by code. One way to handle this is that it can be controlled by the test configuration. For databases, if you have key-based database sharding in place, then using any particular keywords in `I'd like TEST` helps you store test data in a different database. This way, we will not mix the production and test data together in the same data base while testing on the production environment. Although, it is not recommended to test on a production environment. But there could be a case where you can not avoid sanity testing on production. You may wonder, is this the correct way or should any in-memory database be invoked during tests? Should there be a real network call? Should a deployed artifact have any loophole code for testing in production? And you may have more such doubts.

Including all of this, our main problem is still the same. Any call to a database or third-party is time consuming. Let's assume that we create these test clones and databases in memory. The first and the utmost benefit is that there will be no network calls and there will be a reduction in the time taken to run the test case. Imagine that you have thousands of test cases or selenium UI test cases. They can take 5 to 6 hours to run depending on the number and complexity of test cases, also test cases, running time will increase if they are communicating over a network and adding data in the database. Doing this will make test cases run faster and the build complexity of components can be at ease. To reduce the headache of network calls, there are prebuilt libraries to solve the problem; for example, we have the inproctester for JVM and plasma for .NET. This way, we can make component testing even closer to reality. The same can be done with a database; any in-memory database such as H2 can help with this. Other than that, any database that has embedded versions such as Elasticsearch and Neo4j can help in this context.

Making all these arrangements enables microservice testing in an isolated environment. This also gives us much more control over the testing environment and makes it easy to replicate the issue reported to occur in production. We can put a hack in the code for requests such as a certain pattern found in the request that should be served by an in-memory test clone, and which patterns should go to in-memory database clones, that should be handled by some configurations outside the application.

Contract testing

A contract is an agreement that explains the way in which two parties deal. In the same way, a contract in the context of services is an agreed format in which a service should be called. Alternatively, in the microservice context, we can say that it is an agreement between a consumer and an API provider that describes the exact manner of communication and the format of the expected input and output. Contract testing is different as compared to integration testing. Let's understand this with an example. Suppose we have a loan servicing application. Before sanctioning a loan to the customer, our application needs to deal with third-party service to know the existing loan amount for that customer, which they might have taken from another bank or a lending company. What happens if, for some reason, someone changes the contract on the third-party service side? Our application integration test works perfectly fine, as we have mocked the external service, but in the production environment, it will fail. This is where contract testing comes into the picture.

All services should be called one by one, independently, and their responses should be matched with the contracts. It should match only input and output contracts without worrying about internal code. For any dependency service, they should be mocked like other testing techniques. For contract testing, one has to design the test case for the API provider, run test cases against it, and store the result somewhere. Now, the related test cases should be run on the client/consumer side and then the input given to the provider service and the output coming from the consumer service should match. Alternatively, it can be done in reverse; first, the test case can be run on the client side and its output can be saved, and then this output is given as an input to the provider side. Then, in the same way, match the expected result and the actual result. This makes sure that the contract has not been broken.

Keeping contract testing in mind at the time of development is really helpful in developing the interface, request parameters, and response parameters in a more accurate way. This way, the service developed will be very robust and will always give the expected result to consumers. For example, if we are giving APIs to other services, then the developer should think about it, as it should be extendible in the future by adding new parameters or changing the functionality, which should not break the contract between the service (until some major changes are going to take place in all the services). Contract testing automatically assures that, even after such changes, the contract is not broken. There are some tools to help us out with this, namely, **Pact** tool and Spring Cloud Contract. Both are open source tools and are available on GitHub.

Pact

Consumer drivers contract testing is another word which will catch the attention of microservice architecture developers. As so many services talk to each other, small changes somewhere in any of the microservices can break the contract. So contract testing can help in this, but automation is required here as well. Pact helps us in this area. The following is the GitHub link of the pact service: `https://github.com/realestate-com-au/pact`

How it works is simple. You run the test on the API provider by mocking all its dependencies and store its responses. After this, replay the same responses to the consumer code; the results should match. For JVM-related languages, there is Pact JVM. The other versions are .NET, Ruby, Python, and Swift.

Spring Cloud Contract

For consumer-driven contracts, Spring also has a project to help the user to easily achieve their contract testing, **Spring Cloud Contract** (**SCC**). It basically consists of three major tools:

- SCC verifier tool
- SCC WireMock
- SCC Stub Runner

By default, SCC supports JSON-based stub (WireMock). The verifier should be added on the producer side. You have to define the contract for your URLs. You can keep that in the same repository or have a completely different repository for it. The producer and the consumer both refer to this repository as the common contract. The following is the sample POM file:

```
<dependencyManagement>
    <dependencies>
             <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-contract-dependencies</artifactId>
            <version>1.1.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
```

```
            <artifactId>spring-cloud-starter-contract-verifier</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-contract-maven-plugin</artifactId>
                <version>1.1.1.RELEASE</version>
                   <extensions>true</extensions>
                <configuration>
                    <baseClassForTests>in.packt.example</baseClassForTests>
                </configuration>
            </plugin>
        <plugins>
    </build>
```

For `baseClassForTest`, one can use `regEx` also.

The idea is fairly simple in a Spring contract. Create a contract and an endpoint. Run the test, which automatically generates `stub.jar`. You can run this `stub.jar` file in the local Tomcat and the consumer side can easily call and match the result. Anything that has failed in the integration test or previous steps can be caught here. So, it reduces the overall risk of the system going to production.

End-to-end testing

End-to-end testing mostly refers to testing the whole application, which includes the controller layer, the service layer, the database, and any third-party layer. The purpose of end-to-end testing is to make sure that every component in the system is performing well and in a manner that they are supposed to. No dependency is supposed to mock in end-to-end testing. This testing is supposed to be done as an assumption of the end user, as it is using this application.

End-to-end testing in monolithic tries to cover all the possible scenarios in the application. On the contrary, the word application has very much changed in the context of microservices. In microservice architecture, every microservice itself is a small and independent application. As in a microservice, everything is distributed, so testing the whole application is challenging in itself. There are many questions raised in E2E testing, what to cover, how much to cover, and the dimensions are also diverse in microservices. E2E testing ensures that every component works together in the way it is expected to and as per the requirement.

Some frameworks help to accomplish the objective. **JBehave** is one of the tools to automate functional testing. It promotes **behaviour-driven development** (**BDD**), which is evolved TDD. JBehave takes user stories as input and ensures that the system response is as expected. This type of testing is much larger and complex than other types of testing.

User interface testing is the testing of the highest order, as it tests the system as an end user would use it. Testing of this level must feel like a user trying to interact with the system. All the databases, interfaces, internal, and third-party services must work together seamlessly to produce the expected results.

One step further

In the microservice field, end-to-end testing is not limited to requirement testing. Taking one step ahead, performance testing should be considered in this. As in a microservice architecture, every service runs on different machines and there is an endpoint in all the services that are called by other services. So, there would be a question as to whether load testing of these endpoints should be done on the component level in isolation or the application level.

If load/performance testing is done on the component level, then it will be very helpful to identify which service is performing slowly. In addition to that, different layers' issues, network layer issues, and machine level issues are also highlighted in this testing. This type of testing points out the bottlenecks of performance. It will also help to identify application strength in network latency, packet loss, and scaling policies of components. Performance testing becomes a challenge for testers. Any new release can cause slow performance at the UI level; then, identifying which microservice is slow becomes a challenge. At this level, monitoring plays an important role in the microservice architecture. We will learn about monitoring in `Chapter 9`, *Monitoring and Scaling*.

Summary

Microservice testing in itself is a very challenging task. There is no silver bullet in testing. It depends on many factors: team structure, DevOps capability, the available infrastructure, and so on. The rule, flow, or sequence of different types of testing changes according to these factors. A combination of testing techniques according to your organization's requirements is the most effective way. Automation testing is very important in the microservice architecture. It is another wheel to gain speed in microservice architecture. This will add more work for the QA team, as they have to write and maintain automated test cases. After all this, it doesn't mean that manual testing is not required. For actual user experience or sentimental testing, manual testing will be more helpful. So, a combination of both of these will be helpful and more productive.

7
Deploying Microservices

Benefit of microservice architecture is that it breaks big problems into smaller ones, and those pieces of code can be shipped separately. If anything goes wrong, the impact of that also becomes smaller, which means the whole system doesn't have to bear the downtime. Another thing that matters is the batch size, that is, how fast we are releasing a microservice. It could be 1 month, 2 months, 6 months, and so on. In traditional ways, the release process is not done often. Because of this, developers are less well versed in releasing, which leads to more mistakes. There are many manual steps, such as shutdown, setup/update infrastructure, deployment, restart, and manual tests. The more the number of manual steps, more are the chances of mistakes. The whole release process is more laborious, cumbersome, and time consuming.

If the release is happening after a very long time, then it is likely that something will go wrong among the different components together, such as version conflicts, incompatible components, and so on. However, it is hard to see what change is causing the problem, because there have been a lot of changes. The point is that those problems are discovered too late. We get feedback too late, because we are not relearning regularly.

On the other hand, a vendor like Amazon releases once in a week or some others multiple times in day. So, how frequently we are releasing, impacts the downtime and also reduces the maintenance windows. To maintain the balance between maintenance window and batch size, we have to work on zero downtime.

This chapter covers the following topics:

- Continuous integration
- Continuous delivery
- Configuring tools for CI and CD with our microservices
- Dockerizing our microservices
- Using open source CI tools with our Dockerized microservices

Continuous integration

Business requirements are rapidly changing. To stay in competition, businesses always look to data, market, trends, and so on with different perspectives and try to match with the most trending changes. Business people like to analyze things, try experimenting, and are keen to learn from results and increase the business value as fast as possible.

These ever-changing business requirements have to be pushed fast in the system to measure the impact. Traditional software development cycle could be long and tedious. The core of the problem starts from here. Longer the cycle, more is the number of issues. Each developer gets a copy of the code from the repository. All developers begin at the same starting point and work on adding a new feature. After that, all developers keep on changing and adding the code. Everybody is working on their own task. As the team finishes its work, everyone starts pushing their code to the central repository. Here, the problem starts. Code in the central repository is changed or some method signature is changed. Alternatively, there may be some API changes, tests start failing due to unknown changes done by another teammate in same feature code, there may be issues in merging the code, there may be compile conflict and test conflict, and so on. After that, there starts another cycle to fix these issues, which extends the release cycle a bit more. Things do not end here, before rolling out these new changes. Also, tests regarding the overall performance of the system are required to gain confidence that the software is working efficiently. So, there is much manual QA testing, slow release changes, manual performance test, and so on. Lots of manual work! Phew!

Where has the word "rapid" gone? Think of another situation. A developer made changes locally, and local tests do the unit test. The developer commits the code, and the **continuous Integration** (**CI**) system checks the code and runs the entire suit of test cases on it to confirm that the latest committed code doesn't break the build and make sure there is no compile time issue. This might cause tests to run 1000 times a day. Successful builds are auto deployed to staging where application acceptance test cases are run. After successful pass of test cases, the staging environment built will be ready as a good candidate of production release.

The basic idea of CI is to integrate continuously. A simpler definition of CI, which I have come across, is CI is the practice of testing each change done to your codebase automatically and as early as possible. The simpler the idea looks, the harder it appears to implement in practice.

To implement CI concepts, you need some fantastic tools for testing for UI as well as backend. Here is a list of tools that you can use to start testing UI:

- Jasmine (JavaScript)
- Selenium (UI testing)
- CasperJs (UI testing)
- Cucumber or Lettuce

The advantage of running all tests immediately for every change is that you know right away if something broken. Any failure can be fixed right away. Automated tests are useful if it runs with every change and continuously. CI is the first step towards continuous deployment.

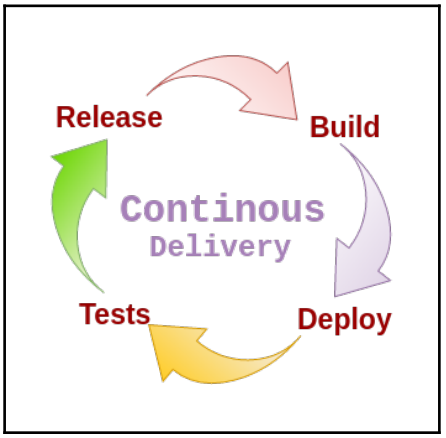
Continuous delivery

We can reduce the risk of things going wrong during release by releasing more often. For this, automating the whole release process (including package release, infrastructure settings, deploy, and final sanity tests) is the only solution. We need to eliminate as many manual steps as we can to increase the release frequency. This phase is **continu0us delivery** (**CD**). Its definition by Wikipedia is as follows:

Continuous delivery (CD) is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time.

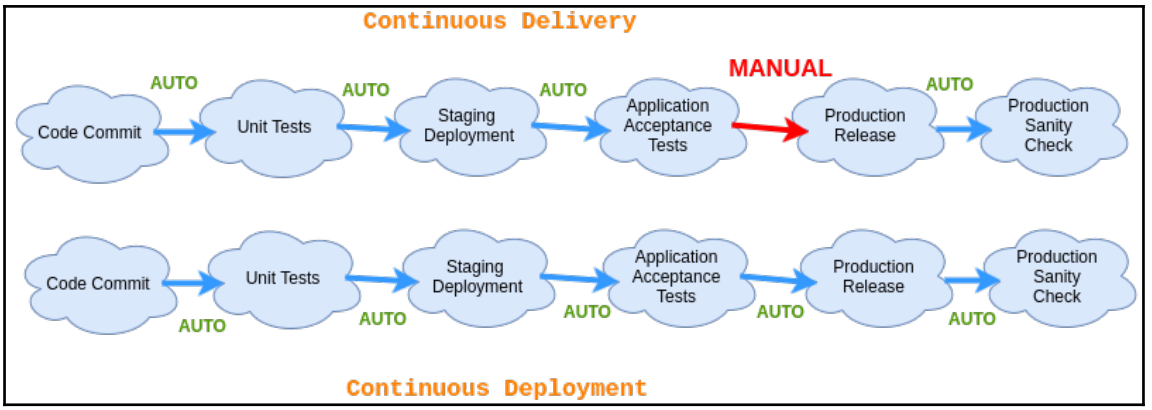
CD is a software strategy that enables any organization to deliver new features to users as fast and as efficiently as possible. The core idea of CD is to create a repeatable, reliable, and incrementally improving process for taking software from developers to real-time customers. We achieve CD by ensuring that code is always in a deployable state, even when thousands of developers are making changes on a daily basis. That's why, CI mostly goes along with CD. If any organization is broken on continuous integration, then it's nearly impossible for them to maintain CD.

The goal of CD is to enable a constant flow of changes into production via an automated software production line. The CD pipeline is what makes it all happen. The idea of having a frequent release sometimes comes with a picture of a less stable release. As a matter of fact, it's not the case. Studies show that the team that is delivering fast is mostly an efficient team and they always deliver good quality release. There are high chances that the team that is not able to deliver fast is not much efficient, which in turn impacts the quality of deliverables. CD is not magic. It's about continuous, daily improvement and discipline of achieving higher performance.



The preceding diagram gives an idea about CD. The image is self-explanatory. Developers build the feature, deploy it to staging , test it, and release it. Transformation between tests and release is a manual process in CD. After testing is done, the QA team normally checks the test results, and then, after satisfaction, they release them to production. So, manual acceptance is still there in CD. CD lets developers automate testing beyond just unit tests so that they can verify application updates across multiple dimensions before deploying to customers.

On the other side, if we make this step also automated, then it becomes another CD. The basic difference between both the CDs is shown in the following diagram:



There is one manual step in continuous delivery, which is recovered in CD. A more confident CD cycle is a good candidate of continuous deployment. It's the next step of continuous delivery. The only way to achieve any CD, continuous delivery or continuous deployment, is automation. Auto-build trigger on code commit, Automation testing, frequent deployment, and so on; are the steps required for CD. Continuous deployment is every technical organization's ultimate goal, but business also has to be ready for it. To wrap up the stuff here, by defining CI and CD practices, and understanding the areas and benefits of both, teams can create, prioritize, and complete tasks in a more efficient and fast manner.

Configuring tools for CI and CD with our microservices

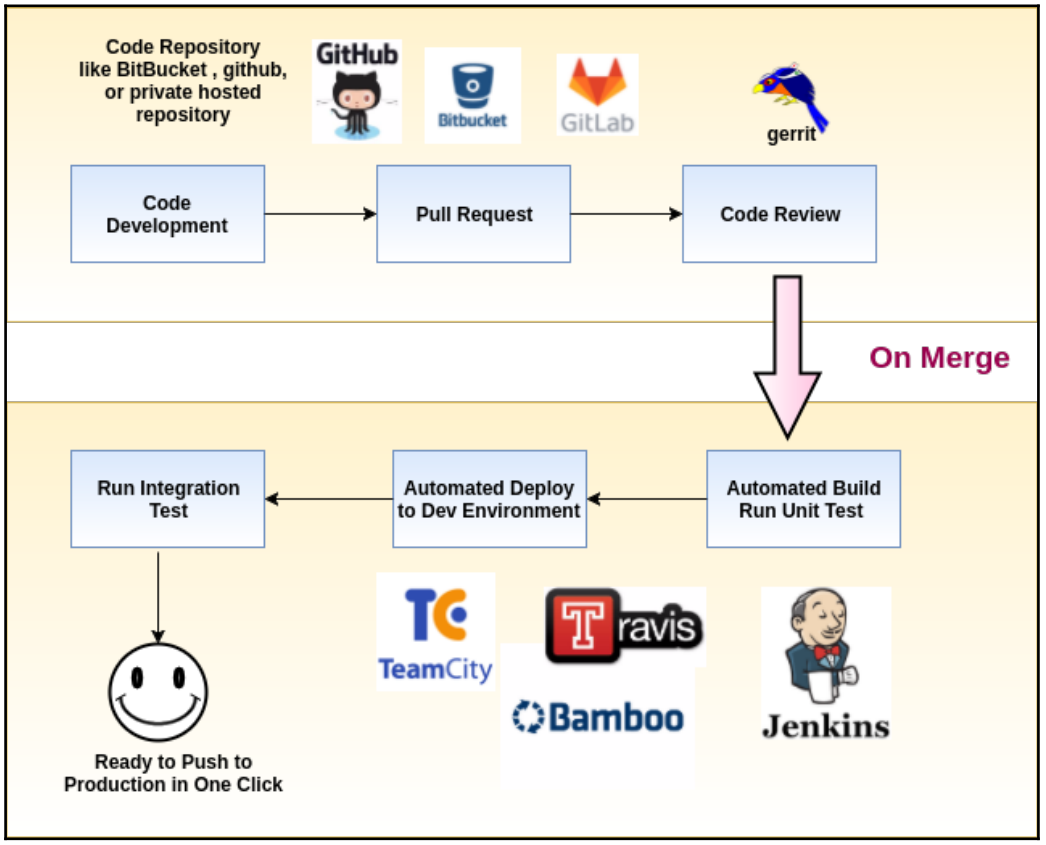
Build, test, and deployment are integral parts of any web project. CI and CD tools are used in the automation of these three steps. To figure out any good tool for CI and CD for your project, make sure that the project supports the preceding three automations.

The upcoming diagram explains the generic work flow. There could be customization in this process, but this is mainly how it works. Let's try to understand its steps:

1. The developer develops and pushes the code.
2. They raise the merge request so that the other member can review the code.

3. The other members review the code of the developer and either pass it or give the developer some review comments. This cycle can happen 2-3 times depending on the length and quality of code.
4. The reviewer merges the code to branch, and it triggers the build of the software package.
5. It is automatically deployed to environment, let's say Dev-environment.
6. All kinds of test cases should be run automatically in the newly formed build.
7. It gives a report of integration tests. If they are successful, the build is ready to hit production.

This cycle can hit 100 times in a day on different components; that's what gives true meaning to complete CI and CD.



Here is a list of tools that are available on the market for CI and CD:

- Jenkins
- TeamCity
- Bamboo
- GitLab (hosted)
- Travis (hosted)
- Codeship (hosted)
- CruiseControl (hosted)
- CircleCI (hosted)

There are many more like them. If we start categorizing them, then we basically find two types of solutions:

- Standalone software solutions
- Hosted solutions

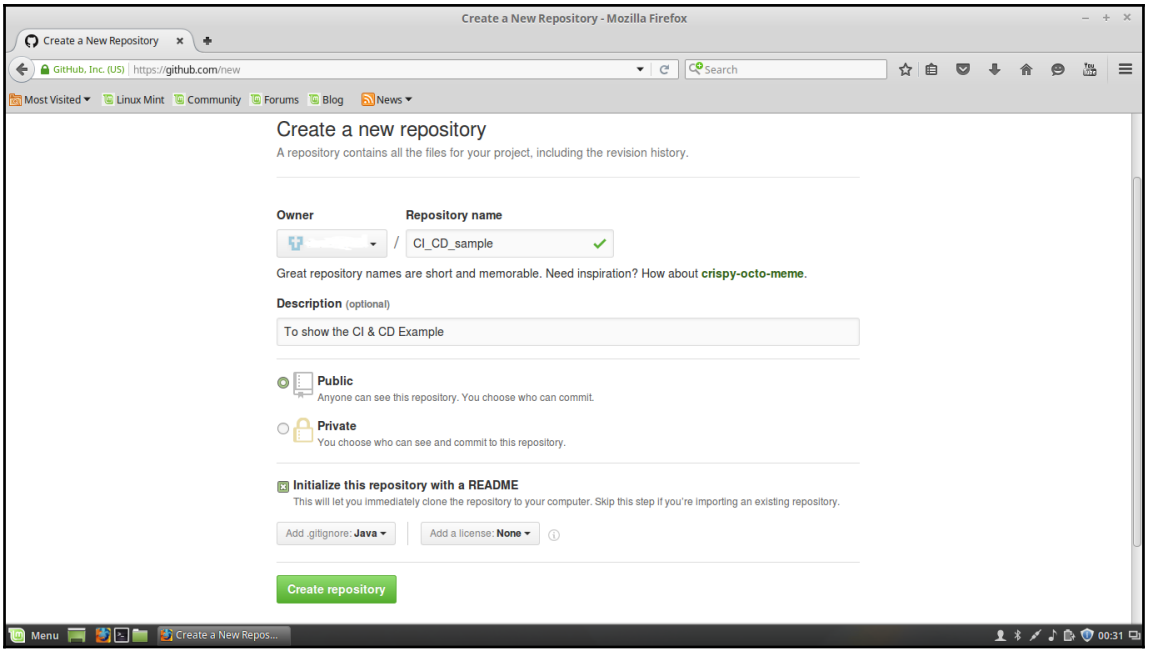
Standalone solutions are installed on your server or even on your laptop, such as Jenkins or Team City, while hosted solutions are like software hosts, somewhere else and you can have your credentials and access them in the browser, such as Codeship, Travis, and so on. The final intent of these is that we should create jobs that perform certain operations such as building, testing, deploying, and so on. Those jobs should be chained together to create a CI/CD pipeline.

Jenkins is the most successful tool among all. The initial success of Jenkins was so big that other products followed its lead and new standalone solutions started coming up. However, Jenkins has already got its grip on the industry. There are over one thousand plugins available in Jenkins. The next generation is pushing CI/CD tools to cloud and are basing themselves on auto-discovery and mostly YML configurations that reside in the same repository as the code that should be moved through the pipeline, such as Bitbucket.

For our microservice, we will use Jenkins, and we can assume our code base on any public repositories, on GitHub or Bitbucket. For this particular example, we will assume our code is on GitHub. With this example, we will try to achieve the following three automations:

- Build automation
- Test automation
- Deployment automation

Let's create a new repository for this example on GitHub and name it as `CI_CD_sample`:



We are making this repo public. We added a `README` file, and for gitignore, we mentioned that it's a Java language project. Clone this project to your local and start creating a sample Spring Boot microservice. We will use Maven as the build script. For this, create a POM file in the root of the project:

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

    <groupId>com.example</groupId>
    <artifactId>CI_CD_sample</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.2.RELEASE</version>
    </parent>
```

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

</project>
```

For Java code, let's write a sample return hello name controller:

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/greet")
    public String greeting(@RequestParam(value="name", required=false,
defaultValue="World") String name, Model model){
        return "Hello "+name+"!";
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(Example.class, args);
    }

}
```

To run this example, run the following command:

```
$ mvn spring-boot:run
```

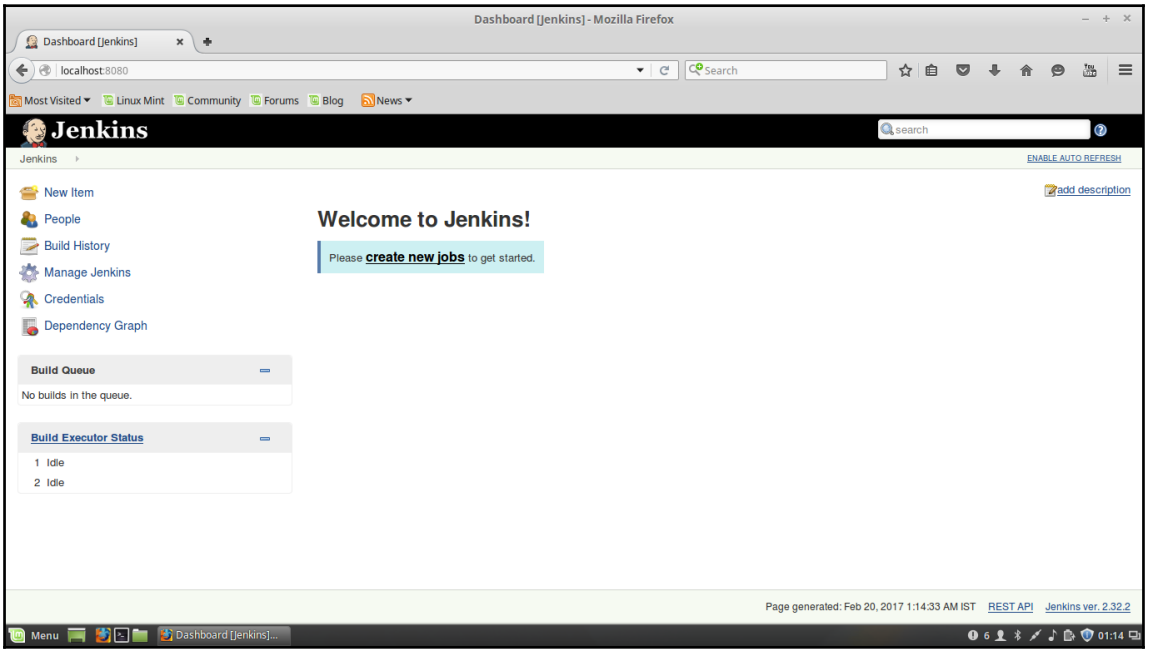
Now, push this code to GitHub, using the following commands:

```
git add .
git commit -m 'Added sample microservice'
git push origin master
```

Either we can use any hosted solution for this, or we can use any standalone software. For now, let's install Jenkins. Installing Jenkins on a Linux-based flavor is pretty easy. You can use the following commands (as per Jenkins, wiki pages):

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key |
sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

After installing, if you run `http://localhost:8080` in the browser, you will see something like this:

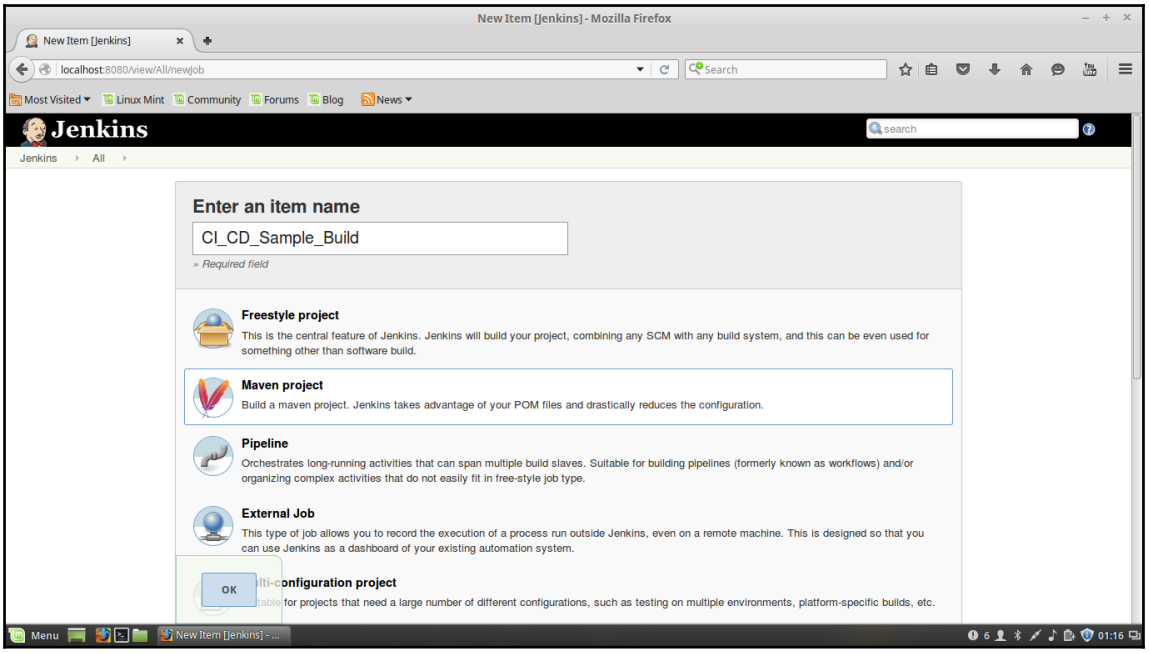


For working with GitHub, add your Jenkins server key in GitHub as the deployment key. On the other hand, in your Jenkins, add new plugins. Under **Manage Jenkins** |**Manage Plugins**, select and install both GitHub and Git plugins. Restart to finish the installation.

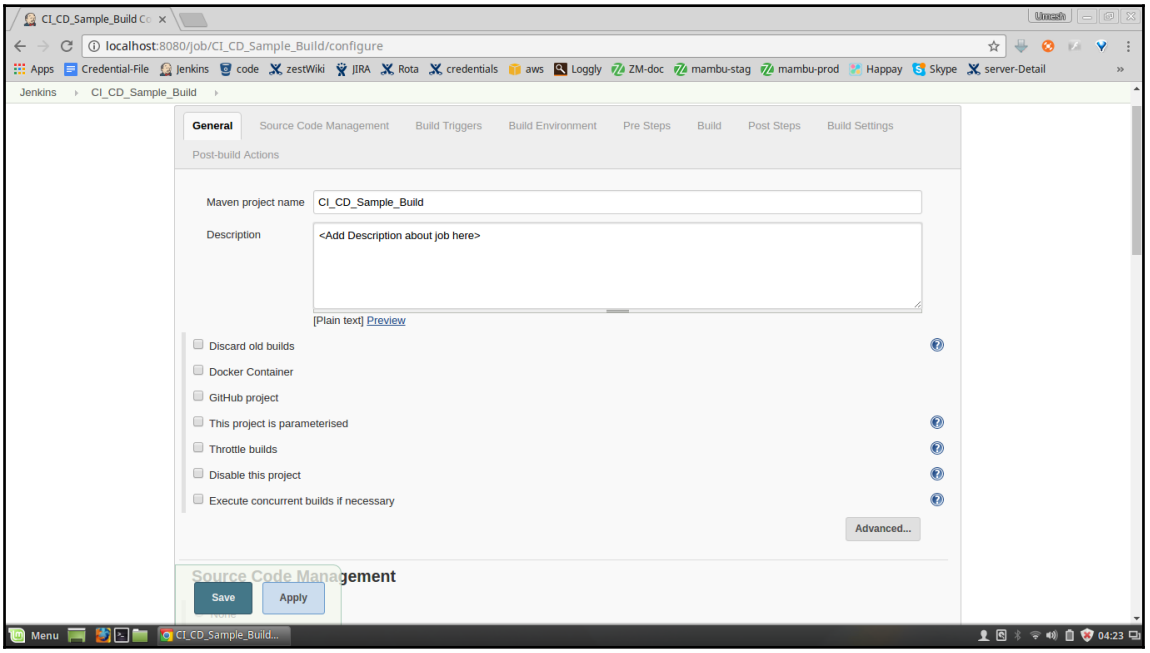
Install the Maven plugin.

Click on **Manage Jenkins** | **Global Tool Configuration** | **Maven installations**, and specify the location of Maven.

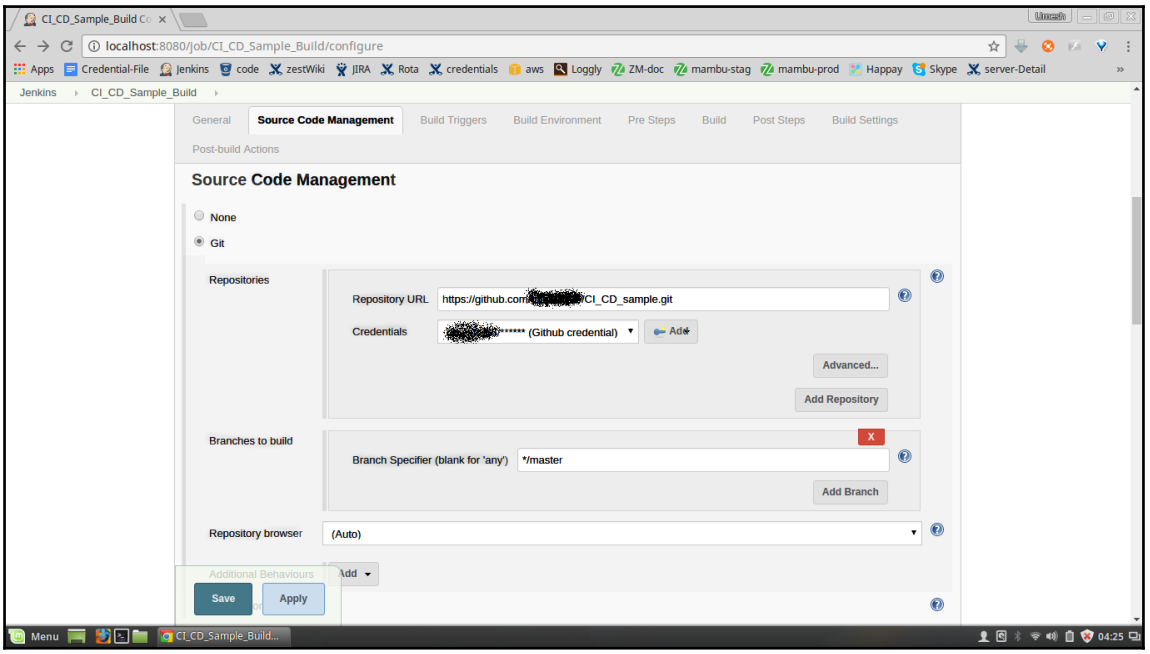
The next step is to create Maven job for build:



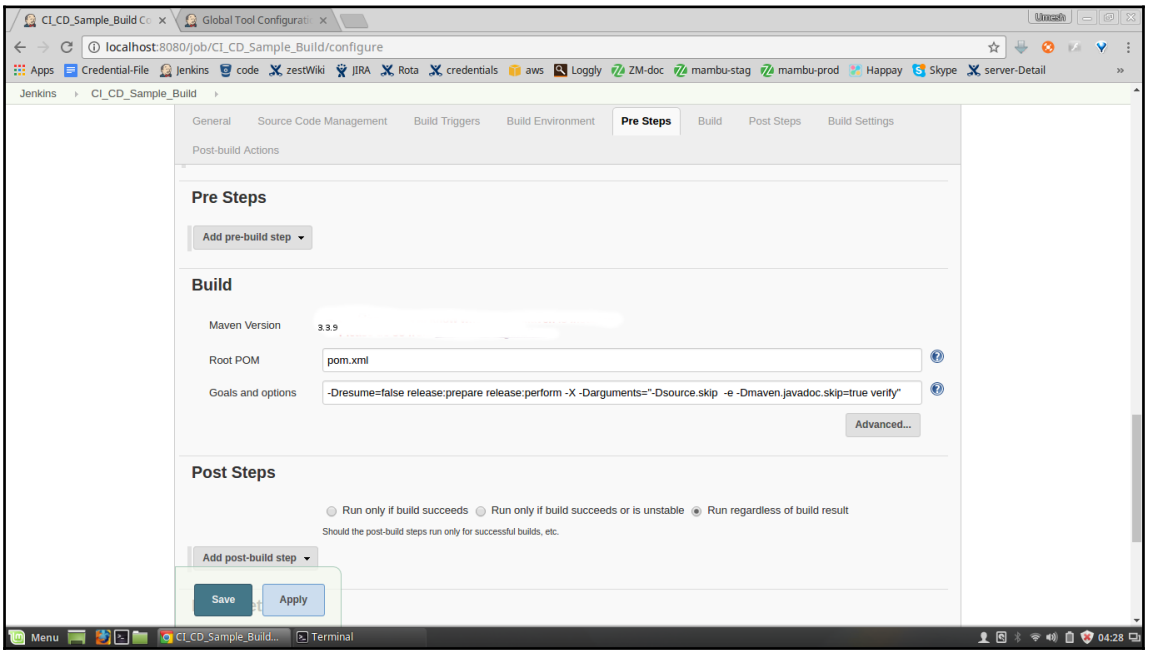
Click on **Create a new job**. Select **Maven project** and give a name to the job. After hitting **OK**, you will get a window to configure the job. Here, you can configure the properties related to the job. In the **Description** box, you can mention the description of the job, such as 'This job is responsible to build module of common component':



The next step is to mention the Git repository form where this job should pick the code to build. As it is a Maven project, Jenkins will look for the `pom.xml` file on the root of the project:



In the **Repository URL** box, mention your own GitHub repository URL. Then, in the **Build** section, you can mention about the goals, such as release perform and so on, that you want to execute on your modules:



For example, in the preceding screenshot, we have mentioned our goal as follows:

-Dresume=false release:prepare release:perform -X -Darguments="-Dsource.skip -e -Dmaven.javadoc.skip=true verify"

Here, we have mentioned to prepare a release and then perform and skip the doc generation from comments and annotation in this build. Prepare release step of Maven command ensures some of the checks in application. It ensures the following:

- There should not be any SNAPSHOT dependencies in the POM file
- There should not be any uncommitted changes in code
- The version is changed in the POMs from x-SNAPSHOT to a new SNAPSHOT-version
- Tag the code in the SCM with a version name (this will be prompted for)
- Commit the new POM file in SCM

Any failure in the preceding steps will result in failure release prepare phase, which results in build failure.

Now, we are ready to run our first Jenkins job with the Maven build.

Dockerizing our microservice

Dockers are gaining more popularity these days. It helps developers create smaller modular components, and also the packaging and distributing of the code is easy in Docker. These are the reasons why, Docker has also become the appropriate choice in microservice platform. We will also be learning to use Docker in a microservice. Before that, let's understand Docker a little bit and learn how to install Docker.

Docker

Container is not a new term; it has been there since long. Containers are evolving during this time. They are lightweight OS virtualization. Containers are run on a host OS, which uses and shares resources from the host kernel. Each container is run in isolation, but they share underlying operating system resources though a layer. Containers come with their own benefits:

- Lightweight
- High performance and speed
- Ship infrastructure with code
- Support microservice architecture
- Lesser portability issues
- Simplifies DevOps

Some organizations have their own version of containers. BSD has a `jail` command which is a type of container they have been using for a while now. Also, Solaris Zones, and Google control groups were already having a flavor of container service , since long a long time. Docker is the most adopted container technology by the community. So, it has given standardization to containers. As it is a container, it inherits all the benefits that are mentioned for containers. Docker is a lightweight layer and sits over the host OS. Initially Docker used the **Linux container components** (**LXC**). However, now, they are on using **runC**. Here are a few main components of Docker.

Docker engine

It is the layer that sits over the host OS. The term Docker engine can be called **warper** for Docker client and Docker daemon. The engine is responsible for orchestrating the containers. End users such as developers or applications interact with the Docker client, which, in turn, pass the commands to Docker daemon. Docker daemon is the one that executes the commands.

Docker images

A Docker image is a read-only template. These images are created, saved, and used later. The Docker files define what an image should contain. These images are used to create Docker containers. It is the basic building component of Docker containers.

Docker storage

It uses a layered filesystem **AuFS** and manages networking. AuFS uses the **copy-on-write** (**CoW**) concept. So, it has a read-only layer and write-only layer, which are merged together after an operation.

How things work in Docker

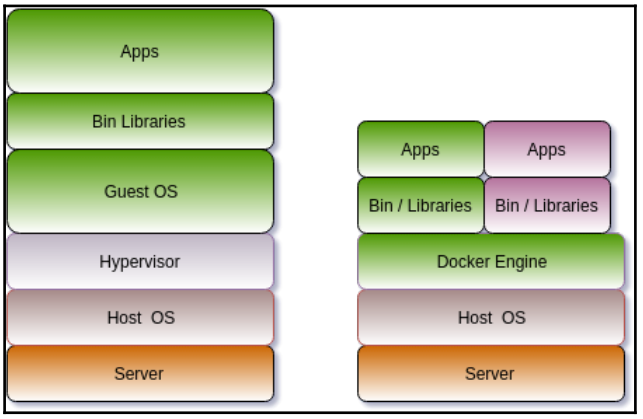
Typically, the developer writes a set of commands in a file, which is a Docker file. The Docker file mentions the base image also, such as Linux, with any web server or something that is required for the application to run. These base images can be found easily on Docker Hub. After executing the instructions mentioned in the Docker file, the Docker image is built. All the prerequisites, such as copying the application's executable code or configurations, are injected in the Docker image. This is also suppose to be mentioned in the Docker file. Then, this image can be shipped to any environment. These images can also be stored in the repository for future use if is it considered to be a base image for future releases.

Public, private, and official image repositories

It is public repository to store the most used Docker images. This ever-evolving repository by community helps a lot in the rising of the Docker. It also provides a private repository to store images. It's the developer's choice to upload their Docker image to the public repository or private repository. Official repositories are organized and promoted by the Docker Hub. These are mostly for OS repositories or languages runtime repositories.

Docker versus VMs

VMs and Dockers work in very different environments. Dockers are lightweight compared to VMs. The main difference between VMs and Docker is VMs provide hardware virtualization and Docker provides operating system virtualization. Let's observe VM's working in more detail. The following diagram describes the difference between VM and Docker internals in terms of layers:



VMs are the images of the operating system that run on top of the host operating system using hypervisor. VMs themselves have a full-fledged operating system bundled in image and required application on top of it. Any hardware request, even taking an input from the user via the keyboard, travels though a running application to the guest operating system call to hypervisor to the host operating system to hardware. A VM operating system doesn't call hardware directly. This causes a delay in response. One VM can be GBs in size. This can result in heavy loading at the start. If the number of VMs increases from one to three on the same machine, this could kill the speed. With an increasing number of VMs, the issue will be the increasing number of OSes running on the same host machine. As there is no provision in the VMs to share guest OS resources among VMs, every VM will be running its own guest OS, which will cost in terms of host machine resources.

Dockers, unlike VMs, share the host system kernel resources among all containers on that machine. In Docker systems, abstraction takes place at user space level. There is no separate kernel load, which results in less consumption in resources such as memory, CPU cycle, and so on. Starting up the full VM will take time in minutes, while Docker starts in seconds. The only limitation in container is it can't run completely different operating systems inside containers, which is possible in VMs. Complete isolation can be achieved in VMs only, but speed, high performance, and a shareable kernel can be achieved in Docker.

Installing Docker in Linux

To run Docker, the minimum kernel version required is 3.13. Check your kernel version, and if it is less than the required version, upgrade it. To install Docker on your machine, follow these steps:

1. Log in to your machine console. Make sure you have sudo/root privilege.
2. Update the `apt- package` information with the following command:

```
sudo apt-get update
```

3. CA certificates need to be installed for ensuring it works on the HTTPS method with the `sudo apt-get install apt-transport-https ca-certificates` command.
4. Add a new **GNU Privacy Guard** (**GPG**) key in Linux. This is used for secure communication between entities:

```
sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

5. Add the Docker entry in the `apt-get` source list. For that, open a file (create if it is not there) `/etc/apt/sources.list.d/docker.list` and add an entry in the `deb https://apt.dockerproject.org/repo ubuntu-xenial main` file (for Ubuntu, 16.04). Save the file after this.
6. Run the `sudo apt-get update` command again.
7. It is recommended that you install `linux-image-extra-*` for having AuFS storage in function.
8. We are good to install the Docker engine now. Run the `sudo apt-get install docker-engine` command. This will install the Docker engine in the system.
9. To start the Docker engine as daemon, run the `sudo service docker start` command.

Docker should be installed here and running in the background as a daemon. It can be tested by running the `docker run hello-world` command. This would trigger Docker to look for the hello-world image locally. If it doesn't find this image locally, then Docker will go to the Docker Hub to fetch the image. It will save it locally and then run it from there.

Docker allows you to treat your infrastructure as code. This code is your Docker files. Docker files are composed of various commands or scripts. During the step of deployment, these steps commands are performed one by one. Like any header file in coding, a Docker file also has the `FROM` command. This tells you what the base image is, from which this image has to be built. That image can be your propriety image or any public image. Let''s see how this Docker file can help us in the build process.

There are a few options available on how to use the `maven-docker` plugin, but we will be working with Spotify. It's more of a personal choice. If you search the `maven-docker` plugin over GitHub, you can find a handy number of plugins. The process is as follows:

1. Create the Maven package.
2. Build Docker image.
3. Push Docker image to Docker Hub.

To go through this process, we need two extra things in our microservice: the `spotify` plugin (and its various properties) in `pom.xml` and the Docker file inside our project. The `spotify` plugin help us run that Docker file to create an image, and the Docker file instructs to run our project inside the image.

You can add the `spotify` plugin in the `pom.xml` file as follows:

```
<pluginManagement>
    <plugins>
        <plugin>
            <groupId>com.spotify</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <version>0.4.11</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>build</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
<dockerDirectory>${project.build.directory}/classes</dockerDirectory>
                <imageName>Demo/${project.artifactId}</imageName>
```

```
                <resources>
                    <resource>
                        <targetPath>/</targetPath>
                        <directory>${project.basedir}</directory>
                        <excludes>
                            <exclude>target/**/*</exclude>
                            <exclude>pom.xml</exclude>
                            <exclude>*.iml</exclude>
                        </excludes>
                    </resource>
                    <rescource>
                        <targetPath>/</targetPath>
                        <directory>${project.build.directory}</directory>
                        <include>webgate.war</include>
                    </rescource>
                </resources>
            </configuration>
        </plugin>
    </plugins>
</pluginManagement>
```

Let's try to understand the preceding plugin. There are a few segments we can see in the plugin. For example, in the execution phase, we defined the package phase with the goal of build. Let's visit the next important entry in the POM file:

```
<dockerDirectory>${project.build.directory}/classes</dockerDirectory>
```

Here, we have instructed it to look for the Docker file in our modules.

To run this, you have to run the following command:

```
$ mvn package docker:build
```

The `mvn` package is supposed to run first. This will build the image, but will not push it. If you want to push it to any registry, there are two ways. You can use the following line in the configuration segment of the plugin:

```
<pushImage>${push.image}</pushImage>
```

The second option is to give the command-line option. Then, the whole command would be as follows:

```
$ mvn package docker:build –DpushImage
```

By default, this will push to Docker Hub under organization if you have property:

```
 <docker.image.prefix>yourCompany</docker.image.prefix>
```

You have to add this prefix in the `<imageName>` tag:

```
<imageName>${docker.image.prefix}/${project.artifactId}</imageName>
```

This `yourCompany` value should be declared as the organization in the Docker Hub, and you should be part of it; else it will fail. You can also push to your own registry as follows:

```
<docker.registry>docker-registry.alooma.com:5000/</docker.registry>
```

You can change the image name as follows:

```
${docker.registry}Demo/${project.artifactId}
```

One more option to name the image nicely is to append a commit ID also in the image name. There are many plugins (depending on your build tool) you can use to get the last commit ID. You can explore easily:

```
<resources>
    <resource>
        <directory>${project.basedir}</directory>
        <filtering>true</filtering>
        <includes>
            <include>**/Dockerfile</include>
        </includes>
    </resource>
</resources>
```

This is the `<resource>` tag that instructs Maven to include the Docker file for the module. This Docker file helps create an image. The Docker file should reside in here and look like this:

```
FROM maven:3.3-jdk-8-onbuild
```

If you want to run any specific command in Docker as soon as it comes up, add following line:

```
CMD ["java","-jar","/usr/app.jar"]
```

Now, our service is well Dockerized and can be used. Let's configure CI tools to complete the pipeline.

Using open source CI tools with our Dockerized microservices

Microservice always promotes fast shipping and deployment for any new feature or code base. To make this happen, we have to configure our CI and CD environments according to that. Dockerizing our microservice also requires our CI and CD environments to accommodate fast shipping and deployment. In this section, we will try to demonstrate how to build a pipeline for a containerized microservice with the help of Jenkins, Maven, and Docker composers. We will build the Docker image of our microservice, push it to the repository, and then run the Docker container with image. Here is the cycle we will follow in this:

- The developer pushes code to GitHub and GitHub hook notifies the Jenkins job
- Jenkins pulls the repository code from GitHub, including the Docker file committed by the developer
- Jenkins compiles the source code, runs unit test cases, and then builds and packages with the help of `maven-docker`
- Deploy the last successful build artifacts
- Build a Docker image with Docker composer and run tests on it
- On successful completion of the previous step, publish the image to Docker repositories

Integrating Docker into their CI pipeline helps accelerate the efficiency of the build cycle, reduce job time, increase the volume of jobs run, remove restrictions over language stacks, and improve overall infrastructure utilization. Docker containers are portable across environments, whether they are developer laptops or production machines.

Once the containers have been successfully built and configured, we will run a series of integration tests to confirm whether the services are up and running. After a successful test happens, the image will move to the Docker Hub registry. One thing we have to mention here is that we are having an assumption that the Docker file will be committed in a version-controlled system. For the sake of example here, we are taking our **Version Control System** (**VCS**) as GitHub, as it gives many already built facilities. One can set up the trigger to build in your Jenkins via Webhooks in GitHub. This is done via the **Webhooks and Services** section of the **Settings** page.

In the example, we are assuming a setup of the Jenkins master-and-slave architecture. The GitHub plugin needs to be installed on the Jenkins master. This plugin keeps pooling the designated GitHub repository and whenever it finds a new change pushed to this repository it triggers a Jenkins job. Jenkins slave will run all tests over the image builds inside Docker containers. The slave should have a Java runtime environment, SSHD enabled, and should have a Docker engine running.

Create a job on Jenkins and restrict it to run on newly created Jenkins slave. The job should have a build trigger as GitHub commit. There will be an option of **Poll SCM**; make sure to check that. This option works with Webhooks you have given on GitHub repositories. After doing this, we are good to write the job shell script. In executing the shell section, write the following lines:

```
# build docker image
docker build --pull=true -t <Your_Repository_Path>:$GIT_COMMIT

 # test docker image
docker run -i --rm <Your_Repository_Path>:$GIT_COMMIT ./script/test
# push docker image
docker push <Your_Repository_Path>:$GIT_COMMIT
```

Let's understand the preceding three lines. The following line is responsible for making a Docker build:

```
# build docker image
docker build --pull=true -t <Your_Repository_Path>:$GIT_COMMIT
```

- `docker`: It is required to mention to the Docker engine that its command for Docker engine
- `build`: This is a command that initiates the build of an image

Options:

- `--pull`: It means it always attempts to pull the newer version of an image
- `-t`: It signifies the repository name with and optional tag to image

Here, we are tagging it Git commit ID. The `$GIT_COMMIT` is the environment variable for Jenkins and comes with the GitHub plugin. So, we don't have to care about filling it in.

The second line mentions about running the test cases in the Docker image:

```
# test docker image
docker run -i --rm <Your_Repository_Path>:$GIT_COMMIT ./script/test
```

- `docker` is the same as mentioned earlier.
- `run`: This is the command to create a container and then start it with the given script.

Options:

- `-i`: It is an option of interactive option. It can be `-interactive=false/true`
- `--rm`: It tells the engine automatically remove the container when it exits.
- `./script/test`: It contains the script to run the test. So, the Docker engine will run this script over the build. In this script, we can write and integrate tests also.

The third one is very simple and plain and tells the engine to push your image to the repository:

```
# push docker image
docker push <Your_Repository_Path>:$GIT_COMMIT
```

Summary

In this chapter, we discussed CI and CD and understood what these terms are, why they are important in shipping fast, and how they perfectly match with microservice architecture. To move things fast, ship fast. Moving big chunks is always a painful and a bit risky. So, microservice breaks the release into many and makes them independent. So, any service's new version can be shipped at any time and reverted back any time. To match up with the speed, you need automatic tests on newly integrated code (CI) and the deployment pipeline (CD). We discussed some CI and CD tools available on the market, and then, we tried some examples with Jenkins, Maven, and Spring Boot technologies. We have also introduced the Docker concept and discussed how we can use it with our services.

8
Evolving the Existing System

Up until now, we have heard a lot about microservices. In plain English, we can say that it's a way to break a large system into smaller subsystems in order to try and make the system as a whole work more efficiently. It should be easier to build and maintain. As much as it sounds good, at the same time, it is a challenge to break up any platform into a microservice pattern. It doesn't matter whether you are building it from scratch or migrating from a monolithic to a microservice architecture, it takes a lot of time, effort, expertise, and research to do so.

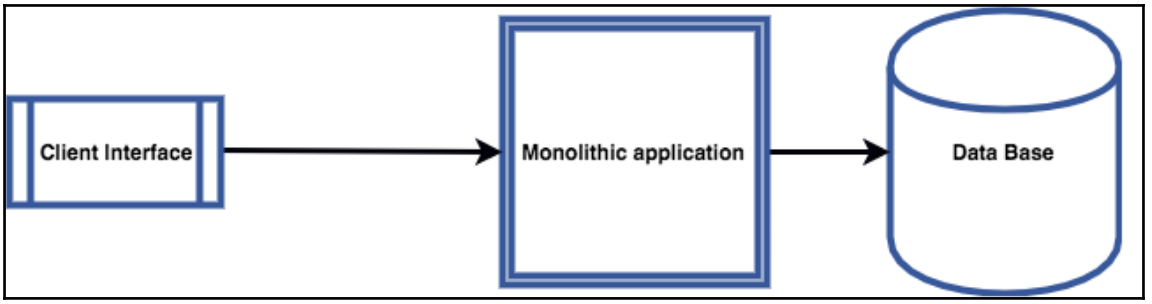
Say you have had a successful monolithic application, running for the last few years: Why would you want to change it? You are using it for a business, and a business needs consumers. If you see an increase in the number of internet users, then it makes sense for you to try to increase sales by increasing your digital presence and attracting more and more traffic. With good digital marketing, the number of visitors to your site will increase, and sooner or later your website will reach the point where the response time will start dipping. If you have an e-commerce platform and it is popular, then one big sales day can be a hard time for your platform. So, whichever pattern or technology was good yesterday for your platform, will not necessarily be best for the future, because of the increase in traffic. Something has to change to accommodate the increase in the number of requests. The microservice architecture may be the best option for you--it depends on the specific details of your platform.

Another key factor is dependencies between the features of your platform. A change in one feature that is seemingly irrelevant to another might cause a problem that means you have to deploy the whole application again, which means downtime for the whole application. If there is a failure in an unrelated feature, it can cause the whole system to experience downtime, even if the feature is not used by the user, or is obsolete.

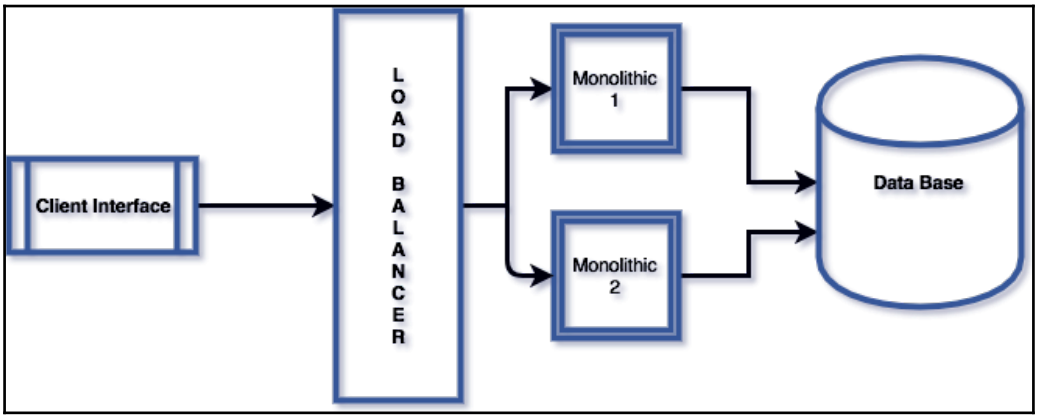
In this chapter, we will try to address these challenges, and look at the best practice one can follow. The following topics will be covered in this chapter:

- Primary mechanisms to evolve a microservices architecture
- Example walkthrough on how to evolve the sample application
- Challenges that you may face

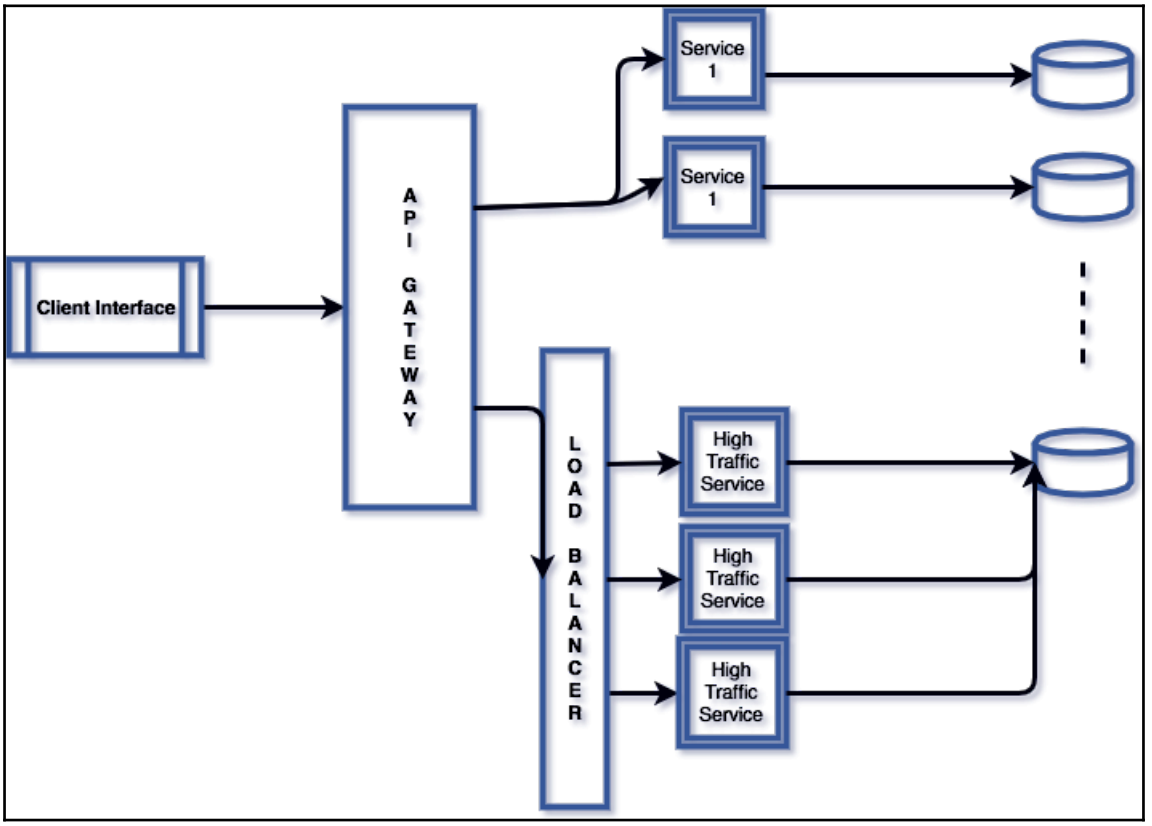
A typical monolithic application consists of a UI layer, controller layer, service layer, persistence layer, and a database. In the very first stage, the client request is directly handled by the monolithic application. After performing some business logic or database interaction, it will be delivered to the client:



Should the traffic increase on your platform, whether for one day or permanently, then one option would be to add multiple servers to your application and add a load balancer in front of those applications:



The problem is that the traffic load for which we have to scale this whole monolithic application won't use all the modules/components. The traffic mostly uses some of the services, but not all. But you still have to scale the whole application, including the unused modules. High traffic only hits the subsystem. If you want to scale only the subsystem (which is facing high traffic) in isolation, then it is not possible in monolithic applications. To scale any subsystem, you have to scale the complete application. In microservice design, you have the option to scale only those services that are facing high traffic, not all the services:



To achieve this pattern, there are a few hurdles to overcome. First of all, you need a technician and a domain expert who understands the domain. These people will understand the domain and try to define the different subsystems with a clear-cut separation of duties or a well-defined bounded context.

The second hurdle would be the mindset of the team. The team should feel ownership about the module or service that they have to develop.

Thirdly, the DevOps team would have lots of overhead, as well as new challenges. Before migrating from a monolithic to a microservice architecture, the DevOps team should be prepared to take on these challenges.

The fourth hurdle concerns monitoring, scaling, and logging. Monitoring and taking action when action is required is not an easy task, for any number of services. We will discuss how to solve monitoring, scaling, and logging problems in the next chapter.

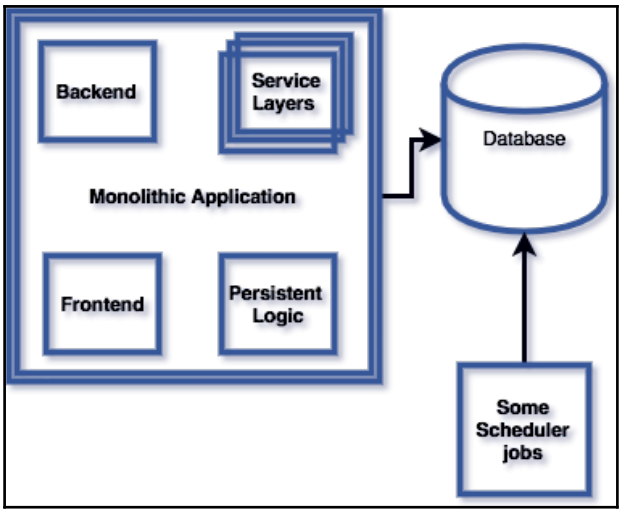
These are the four most important challenges you will face while evolving your system. There are more challenges, such as testing, that we have already discussed, and some of these are going to be discussed in upcoming sections, along with the solutions and recommended best practices to overcome these challenges.

Where to start

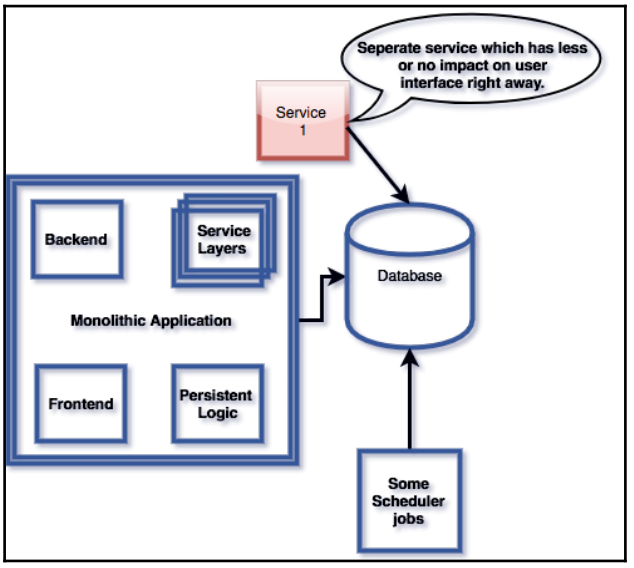
There are two perspectives from which to approach evolving a microservice. One is from an architectural perspective and the other is from a database design perspective. We will discuss both perspectives one by one.

Architectural perspective and best practice

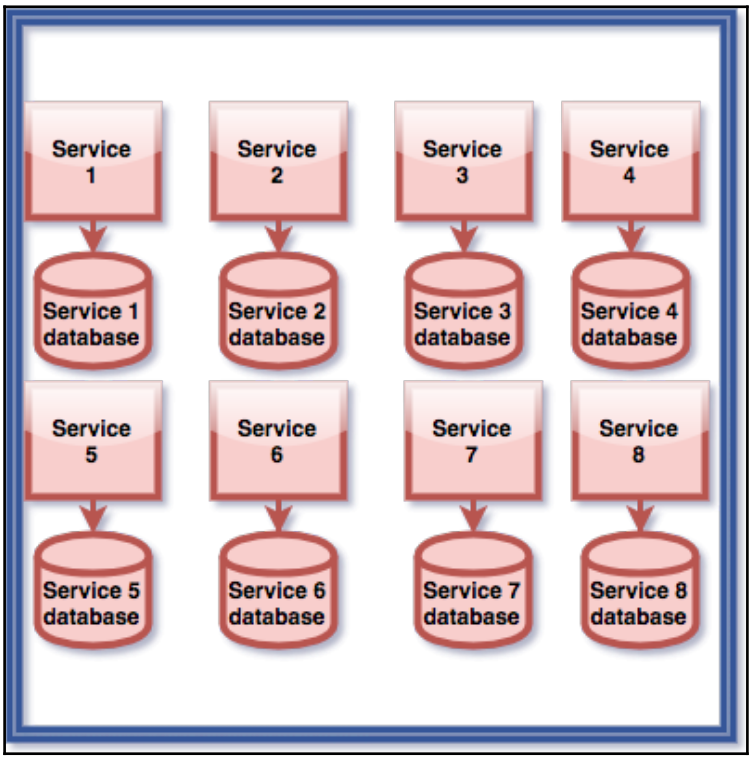
To start with, the best practice is to find out the feature with the lowest performance, or the feature with the least interaction. What would be even better is if you could choose a feature that does not directly interact with the user and is used to doing an offline job. Choose your service very wisely. It should not affect any other feature, and it should be completely independent of other services in terms of code, as well as the deployment cycle. For instance, we could have this service using the same database, but by using a completely independent deployment pipeline:



As in the preceding diagram, there are service layers that refer to lots of the feature's business logic. There could be a schedule jobs service or a feature service. So, after identifying the service that can be taken out as being completely independent, you should create a separate deployment pipeline. This service should talk to the same database, but run separately.



Try to break the monolithic architecture at the service layer in a bounded context. Each service should have its own database. By doing this step by step, you can achieve different microservices in the same platform.



While evolving this way, always try to use the least impactful service in the system. When you reach the core of the system or a critical microservice, try to use a parallel system for some time.

This pattern gives you many advantages, along with the freedom to choose the technology. But try to limit your technology stack. If you have many technologies there, it creates a different set of problems. Suppose you have a similar functionality or some validations in many services that are written in different languages, such as Java, Python, and so on. You can't make changes in that validity logic, as you cannot roll out new logic until you have implementations in all languages. Eventually, this will hamper the speed of development.

One more tip here is to not try to use new technologies just because you have heard of them somewhere. With a wide choice of available technologies, developers sometimes make this mistake. They try to use new technologies in the service. They may decide to use them by estimating future traffic or future problems that are not currently there. This decision is critical; there should be some accommodation in the system's design to support future problems. But by thinking too much, and adding new tools or technologies, this approach could also lead to a possible failure. For example, say a developer replaces the legacy cache logic with some new fancy cache tool. If they configure it wrongly, then this will, in fact, increase the load on the database, rather than saving the database. Don't use too many fancy patterns when starting; you can cause your own downtime by making things too complicated in the beginning.

Whenever you decide to migrate from monolithic to microservice, make sure that you give high importance to monitoring the key metrics. Metrics are required not only at application level, but also at machine level. At application level, metrics such as request per second, error rate, request serving time, and so on should be monitored. At machine level, CPU usage, RAM usage, network in/out, and so on should be monitored. So, during the migration and until you have a successful running plate from inside your microservice, you should have these metrics on your radar.

Apart from the preceding details, the proper cache and throttle should be the main concerns for the API request. For monolithic applications, you should have a proper firewall in place, and the internal application call should be more of a function call, but when you go into the microservice business then you might have a different endpoint open for the world or your consumer application. Some of your clients, whether unknowingly or because of bad code, can make hundreds of calls per second, which will eventually choke your system, so you should have proper throttle policies in place.

In microservice architecture, we try to implement divide and conquer techniques. We divide a big, bundled business logic into small independent submodules that each serve a part of that business logic. During migration, it may be the case that a part of the application is dependent on the legacy application and the microservices. In this situation, we usually create a temporary microservice with the goal of replacing this temporary service with a fully functionally one. Also, as you have lots of communication between the microservice and your microservice to the outside world, proper authentication and authorization should be in place.

Database perspective and best practice

The main challenge of migrating from monolithic to microservice is dividing the database with a clear-cut separation. As mentioned before, you need a domain expert here. Only a domain expert will have the knowledge about which data normally changes together. With experience, you can also have this knowledge. With this knowledge, divide the database in a bounded context.

To start with, rather than actual separation, one can start with database views. By doing this, one can ensure that any separation is working fine. For a new microservice that is separated out, you can read/write from these views. You might feel some performance issues, but this is acceptable until you are in a trial period. Another way of doing it would be to migrate data from the core database to a new database used by the new microservice. This data transfer can be done in two different ways:

- Trigger
- Event sourcing

You can implement the trigger in the core database, and as data is modified in your database and its respective tables, you can send data to a new database, or you can establish a database link to do this. Another non-traditional method is using events. In event sourcing, there are tools that record the event in your current database and replay this event in your new database. Debezium is one of the many open source tools that records the event and can play it on a newly created microservice.

During database migration from a monolithic to a microservice architecture, keep in mind backward compatibility as well as forward compatibility. You should not break something that's already running. You should make sure that everything is set up so that nothing can be shut down in one shot. If you want to delete a column from a database, rename it first and release it. If everything is correctly set up, then you can remove it from the next release. Any migration happening on a microservices database should not be manual. The proper tools should be used for database migration; FlyBase and Liquibase are two such tools that will help you to upgrade your database automatically. Both are supported by many languages, and they also have Maven plugins.

This changeover cannot happen in one day. It is a continuous process until you have completely moved to a microservice architecture, so until this process has finished, try to avoid writing a destructive statement in the database. You can lose data by doing this.

Sample application and its evolution

In this section, we will try to evolve our sample as a microservice application. We will be using the example of an online e-commerce website that sells clothing, accessories, shoes, and so on. This website has started gaining popularity, and they have to show offers on a weekly basis, or sometimes a daily basis, on the website. New features are added daily to the website.

The assumption here is that we are dealing with a single-page application. If we have to evolve this application as a microservice pattern, first we will think of it in terms of its service. There are two major categories of service-view service and backend service. Our first step would be to differentiate the view and backend (model and controller) sections.

The view service is the part that the user sees on the pages. It can be divided again into two parts--static content and dynamic content. Static content will be CSS files, some common JS files, and so on. Dynamic content is the template (consisting of HTML) based on user action. If the user selects the shoes category, and the shoes category has the capability of showing a 3D images of shoes to the user, then the new template would be fetched and added to the current page. This can include the required JS file and the template. So the template would have to include the JS statement, but the JS will be served by a static content server. With the view service, the template service can have more microservices as required, and each microservice can have its own database. The view of the page can be divided into four segments-header segment, offer segment, categories and catalogue segment, and footer segment. Each segment can be served by its own microservice, which again depends on the complexity of the logic you want to build.

The first profit we get by separating the frontend code and backend code is you can make changes to the UI code without having downtime on backend services. You can change header images or offer images on the fly. You can also have a scheduler or location finder service in place. So for one city you can show some offers, and for another city you can show different offers, along with the appropriate images. You can also rotate the offers on a timely basis. If this logic is becoming more complex, then each segment of the pages can have different subteams for the ownership of that segment.

For the backend, if we were developing a monolithic application, we might think of having code together in one big mash and having lots of function calls here and there. But in a microservice pattern, we have to define a clear separation of duty or bounded context. Initially, we can think of a minimum of nine microservices that have different bounded contexts and separations of duty:

- User management service
- Cart/order service

- Categories and cataloger service
- Offers services
- Payment service
- Shipping/tracking service
- Communication service
- Recommendation service
- Scheduler services

All of the preceding services have a different role, and each has its own database.

User management service

The user management service is supposed to perform user-specific operations. These can include login, logout, organizing user details, user address, user preferences about payment, and so on. Login/logout can be taken as a separate service, such as an authentication service. Here, we are talking about a hypothetical service, so we are adding two services together. The reader can make their own decision to turn one service into two when they deal with their own services. However, we would like to recommend here that, until you have a problem with latency or performance, don't break your services further.

Cart/order service

The cart/order service is used to maintain the cart operation and deals with the deletion or addition of items, in much the same way that the categories and cataloger service is used to show the available categories and the different products under those categories, as well as their catalogs. The offers service is used to store offer-related information, such as which offer is live, and in which geolocation. The offers service can have a validation service as well. For instance, it can validate whether one offer should be used twice on the cart, or if the offer that the user applied is still live or not, or applicable to a certain category, and so on. There could be many, many different combinations of these offers and their validity, so the developer could think of having the validation service as separate from the offer service. But, for now, this is not required in our example.

Payment service

The payment service, as the name suggests, will be responsible for collecting payment. It can offer a different way for the user to pay for his purchase, such as a bank transfer, credit card, cash, or PayPal. We can go one step ahead here and can have a wrapper service that will actually check if the current integrated payment gateway is up or down. If the current payment gateway is down, it will use an alternative option of payment gateway, if one is available. Collecting money is the main point of business; you cannot afford to lose the user here, so you need as much fallback as you can have here.

Shipping/tracking and communication service

After the payment service, there are two microservices that start their work. One is the communicator service, which actually shoots an email or SMS to the user seeking an order confirmation, and then, once the order has been confirmed, another email with their order details and an estimated date of arrival. The shipping service is responsible for initiating the shipping process. The same microservice should be responsible for tracking the shipped product.

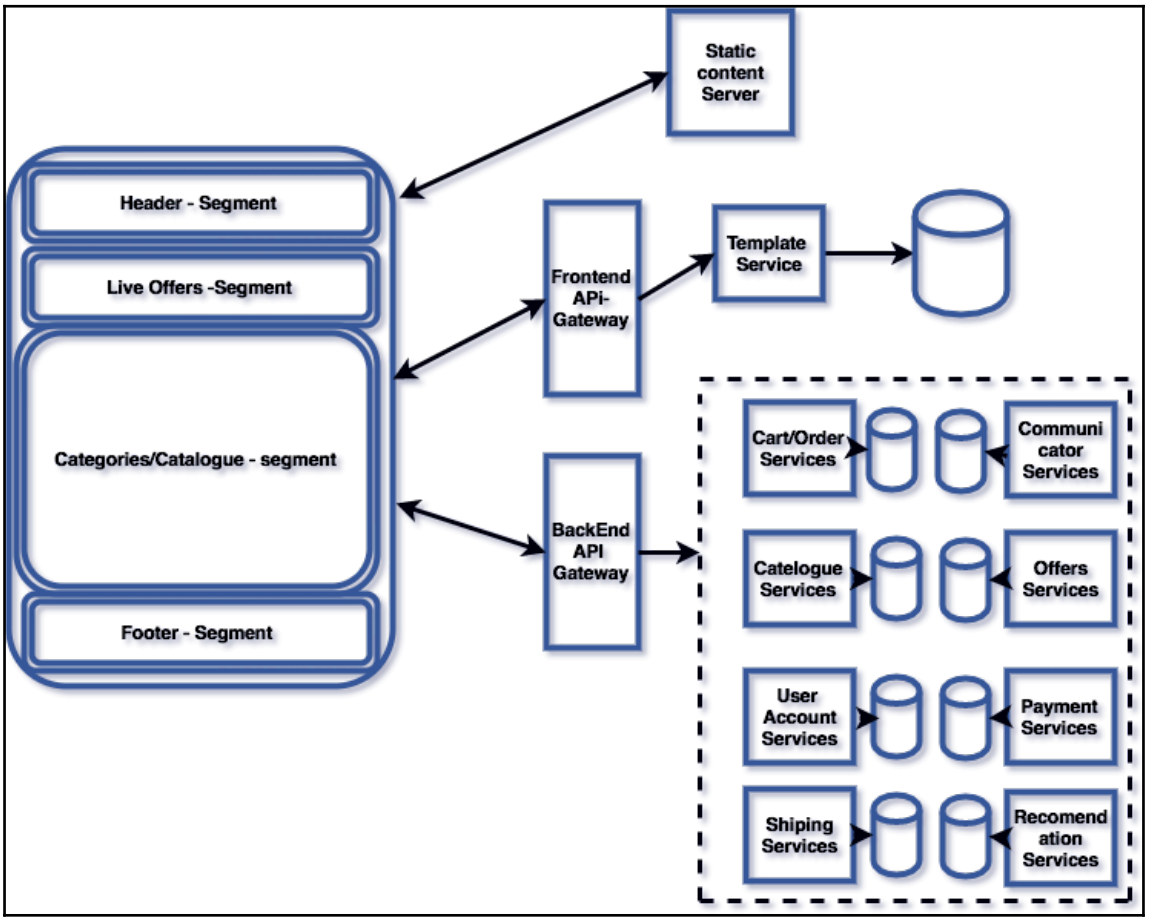
Recommendation service

After finishing the purchase, the user is supposed to be redirected to a page where he/she should be recommended different products that are similar or commonly bought with the product that the user just bought.

Scheduler service

The scheduler service is a service that is supposed to do backend jobs, such as verifying that the payment has been received, reading or transferring the data for different reports for management, and so on.

Each microservice here has its own database, and each microservice can have a different microservice. This makes our application technology agnostic. This is easy if we are thinking from scratch. If we have to migrate it from a monolithic application to a microservice application, then the appropriate method is to take the scheduler and communicator service and separate them out to try to run these roles separately. After completing this task successfully, you can think of taking the catalog service out, and so on. The final picture of your application would be something like the following:



Summary

Migrating from a monolithic to a microservice architecture is always a hard task. Always start with baby steps. Try to take out those services that do not interact directly with the user. Keep an eye on your metrics machine, as well as the application-level metrics. Try to avoid any destructive statements on the database. Use new columns, and refer to the old columns as the dead columns in the database. You must use a domain expert to divide your application into different services with bounded contexts. If this separation isn't good enough and is improperly planned, this will make your microservices fairly complicated in the future. Monitoring and scaling are the next important tasks that we have to do, which we will discuss in the upcoming chapters.

9
Monitoring and Scaling

Moving to microservice architecture from a monolithic application gives a number of benefits to the company, such as a smaller code base, fast shipping, adding new features quickly, easy to develop, easy to scale certain parts of application, and so on. However, with the growing use of microservice architecture, their complexity also increases. It becomes very challenging to know what is happening inside the system. The flowing of so many messages inside the system makes it hard to know the exact state of the system. Also, dynamic architectures and the auto-scale world makes things more complex. In this chapter, we will take a look at some of the key points to keep in mind for monitoring and scaling microservices.

This chapter covers the following topics:

- Principles in monitoring a microservices system
- Principles in scaling a microservices system
- Practical options for both monitoring and scaling microservices

Principles in monitoring a microservice system

Monitoring is one of the key considerations in microservice control systems. Unfortunately, migrating to a microservices architecture gives you flexibility, but it has its challenges as well, since it turns one single monolithic application to complex distributed micro systems. How many nodes or microservices you have is directly proportional to the chances of failure. It can be extremely difficult to have visibility about the communication and request flow between different microservices. In this architecture, monitoring becomes as important as development. Any application, whether it is a web application or mobile application, might have a single API call to system, but behind the scenes, that single API call might turn into tons of internal communication. If anything fails here, even one of them, DevOps and developer have to track it back. It becomes very important for developers and DevOps to have visibility of the internal system so that they can take remedy steps quickly. At any failure situation, if we have tones of megabytes of logs, it will become hard and time consuming to identify the problem. So, the right kind of tools are necessary to monitor such a large number of services, which, in turn, helps the team point out the exact problem quickly.

There are some best practices identified in the monitoring system of microservices, which we will discuss.

How and who will see the alerts

When we talked about developing microservices with loosely coupled services in that architecture, the development team of that particular service takes end-to-end ownership of the service. So, it makes more sense to give them power to see and design their alert metrics for that service. Their own designed metrics gives them real and meaningful alerts. If the metrics are decided on and designed by the DevOps team, then they might not do justice with true visualization of what is happening inside the service. So, set up your monitoring platform in such a way that allows every microservices team to decide and design their alerts, metrics, and dashboards. Also, the DevOps team should be able to see the overall picture of the system.

Monitoring and channeling from the start

Monitoring should not be noise. Sometimes, teams design so many alerts in place without giving proper thought to them. Some are important, and the rest are just noise. Developers start ignoring the noise, and in between, they miss the important alerts as well. So, think about monitoring from the beginning. Create only those alerts that matter. Put justified effort into planning the alerts on every service. Developers should be involved in it (as per the previous point). Also, the channel is important. Something such as emails are very old, and people mostly create filters (if a platform is too noisy). Rather than that, there should be a messaging channel such as slack or so on. Going one step ahead, it should assign a task to the owner of the microservice. Continuously improve your alerts, stop the noise, and add the important alerts, which you will learn over a period of time.

Autoscale and autodiscovery

In this era of autoscaling and auto services discovery, services are coming, scaling up, and scaling down, as per the need. It all happens very dynamically. Your monitoring platform should be able to show the current number of machines running on a particular service and also show all predefined metrics for those machines, such as CPU load, memory load, traffic load, and so on. It should be dynamically added and removed. Some sort of history metrics should also be stored to know the peak traffic time, how many machines were up and servicing at what capacity. This helps the developer to tune code or autoscale parameters. The challenge this type of system will bring is, at high scale, this could result in running hundreds of agent services and eating up resources. Instrumenting the code inside a service itself will save a lot of headache for the operations teams.

Complexity, operation overhead, and monitoring work increase when we take microservice into the container's world. Containers have lots of promising features to make them a good choice for microservices. At scale, an orchestration service continuously spins up new containers as needed, which make challenges more interesting for DevOps teams to manually define which service should be accommodated in which container. So, to operate in dynamic and container-native environments, your monitoring system should be capable of spanning different locations, such as your AWS CloudWatch and your private data centers.

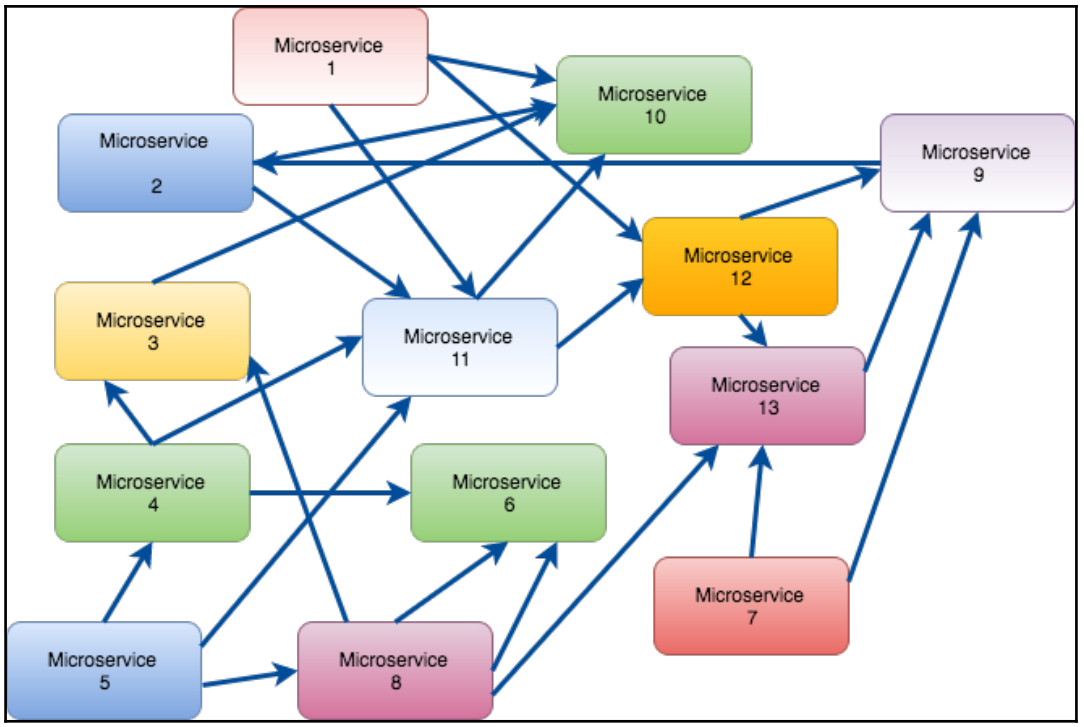
Monitor front door

Traffic on the front door should be monitored carefully, and its metrics should be defined after giving proper thought to it. Front door implies any entry point exposed to the world. It should be HTTP API, or sockets or static web pages, and so on. Deploying public or partner-accessible microservices without protection is lunacy and a recipe for disaster, unless you like failing. Think about the scenario when someone decides to hit your endpoints 10x more than you planned for; sleepless nights for sure. So, monitor microservices that you publish and limit access to them.

The monitoring system should be capable of making you understand the things happening from outside the world, such as a sudden rise and fall in API calls while there are not many active users in the system. It is better if the monitoring system evolves by itself by learning the metrics it collects. One of the major factors is that the monitoring system should be able to show you the data flow and data about service performance. Developers need performance metrics to improve the code in a new version. Performance metrics includes CPU usage, memory usage, time taken to talk to a database or another service, the serving time of one request, response time versus number of request graphs, and so on.

Changing face of monitoring

Monitoring refer to some graphs, alarms, or stats that have to be read by a human. These alerts can be received through email or SMS, or can also be visual on a browser. After observing these alerts, one is supposed to take action. So, raising an alert is automatic, but to rectify it, you need manual intervention. Somewhat like Nagios to perhaps send an alert if something is not working as expected. This is not the case anymore in a microservice environment. With the growing number of microservices in one system, you need smart monitoring in place. If they identity a failure or malfunction in the system, the monitoring system should be smart enough to take corrective action. If they find a service down when it is supposed to be up, then the monitoring system should try to restart that service. If it fails again and again, then it can analyze the log and find some common issues, such as connection time, out or so on. Take predefined action based on log analysis:



In any distributed microservices architecture, communication will be as shown in the preceding diagram. So, there are many services talking to so many other services. So, there are many challenges to maintain many metrics. Metrics such as request per second, available memory, number of live connections, failed tried authentication or expired tokens, how many dropout on which page, and so on. These parameters are important for understanding the user behavior, application behavior in peak and slow time, and so on. Once monitoring is in place, the operation team will be notified about failure. Finding the exact cause and rectifying it in the preceding distributed environment will take time. Then there comes the reactive monitoring. It is very hard to work with distributed systems without reactive monitoring.

Traditionally, all monitoring tools generally support the needs of the DevOps team. Mostly, they have a weak UI experience, and sometimes these tools have only configuration files. However, in microservices, you're decentralizing the tools and processing out to multiple development teams. The development teams will also include non-operations people such as developers, QA, and product management. With modern UI technologies, it should be really easy for teams that are not in operation, such as developers, to set up and use their own monitoring as a self-service, while still giving the DevOps team an overall picture.

Fast shipping is another good feature in microservices , which adds or removes any feature quickly. Make a parallel release of a new feature and send some traffic on that. Basically, here, we are trying to get user reaction on a new topic. This is what we call A/B testing. After this, the product guys try to improve the application. This will raise the bar for monitoring. Here, we are not trying to find failure in the system; instead, we have a variety of applications and are trying to observe user reaction to a new feature, which, in fact, makes the product guy realize the effectiveness of that new feature.

With big data, data science, and microservices, monitoring microservices runtime stats is required to know not only your application platform, but also your application users. You want to know what your users like and dislike, and how they react. Monitoring is not limited to alerts any more.

Logging helps in monitoring

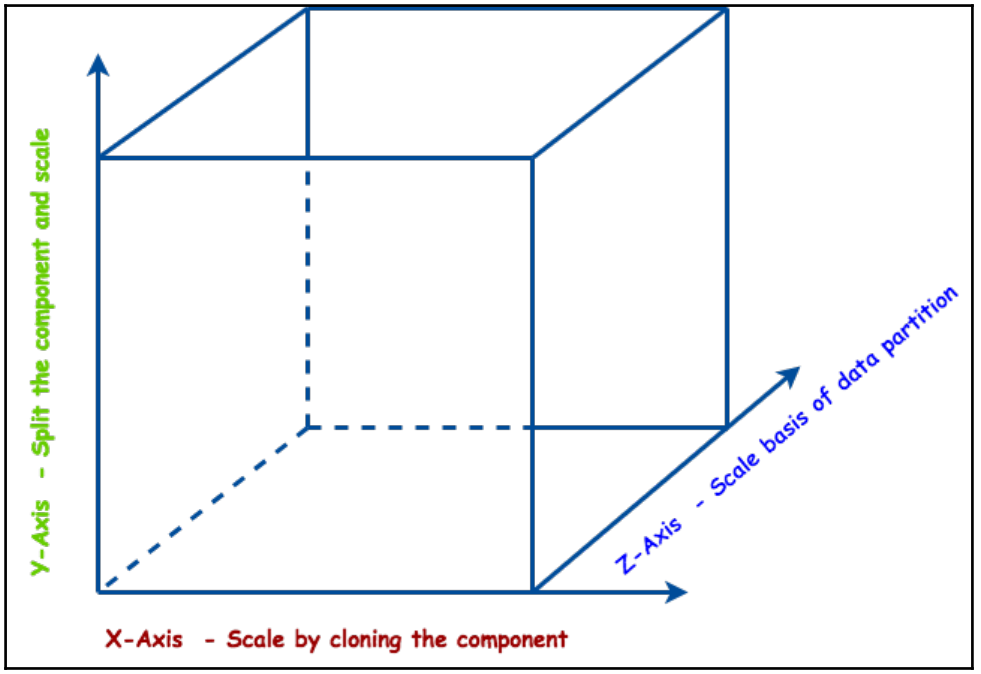
Another challenge if you move from a monolithic application to microservices is the lack of stack traces through services. For one single request, one action is happening in service A, another action is happening in service B, and so on. The final result of that request is a failed response, but what happened inside, which service actually failed, and why are there questions to answer and lots of effort in debugging required? What happens if one of your microservices in your infrastructure starts throwing stack traces? One idea to trace any request is to bind every request with a unique global request ID, which will propagate to every service and log it.

It is good that we can bind different service actions taken for one request with one request ID, but still, the logs may be on different machines. It is still not easy for developers to track down the bug. For that, you might need a tool that will help in centralizing logging windows for developers. There are some tools for that , such as Loggly or ELK, which we will discuss in the tools section.

Principles in scaling a microservices system

One of the features that influence organization design while choosing the microservice architecture is scalability. It gives freedom to developers to think about how their part of the code is going to be scaled and what the strategies could be for that. It's easy in microservices to replicate any particular service with less resources than scaling up the whole monolithic application. With microservices, enterprises are able to flexibly scale at speed and cope with the load variation in the platform.

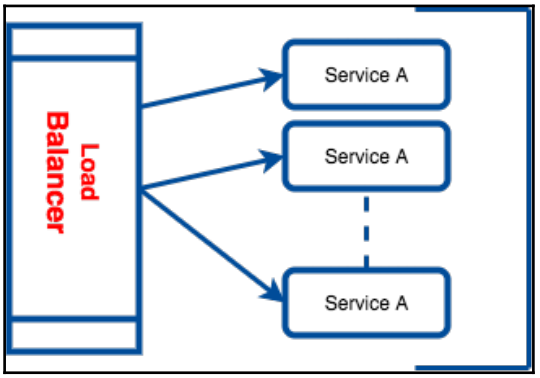
The Art of Scalability, written by Michael T. Fisher and Martin L. Abbott describes a really practical approach to the three-dimension scalability model: the scalecube.



Although the diagram is self-explanatory, to make it a little bit deeper, let's go by each axis.

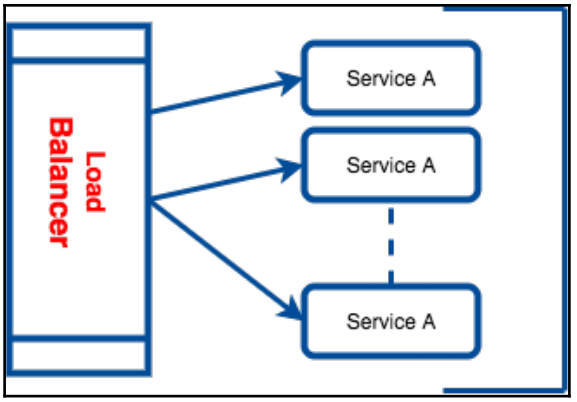
X axis

X axis is the horizontal scale out with adding more machines behind the load balancer. Microservice code is replicated behind the scenes on each machine to handle the traffic. An application doesn't much need to be aware about this type of scaling, typically handled by any TCP load balancer. The problem with this approach is that the application has a state to maintain and it needs lots of work to handle that, also it needs lots of cache work. Performance highly depends on the how cache and its algorithms are performing:



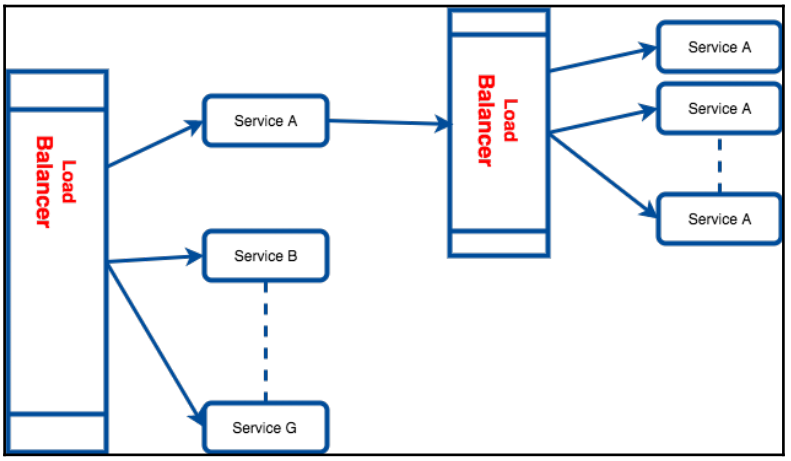
Y axis

It relies on the decomposition of the application into smaller services. It goes well with RESTFul microservice architecture or SOA. It is basically a sharing of application code:

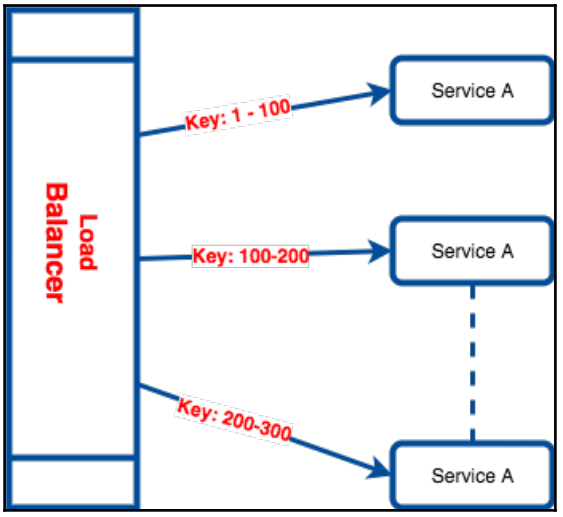


The application is sharded into various small services, and each service has its own bound set of responsibilities to perform. A very common approach to decomposing the application is using verb-based decompositions such as login and checkout, and another approach is based on noun-based decompositions such as customer, vendor, buyer, and so on.

We can also mix the approach of the *x* and *y* axes, as explained in the following diagrams:



Here, services are broken into sub-services first, and then, each service is scaled on the *x* axis basis. The second load balancer runs the clone code for each service. The combination of both *y* and *x* axes scaling is increasingly a good choice.



Z axis

Z axis scaling is the same to some extent as *x* axis scaling, where each server runs an exact copy of the code. The major difference is the request assigned to servers based on some keys. So, actually, each server does not serve the whole data, rather it serves only a subset of data. It is actually a cross between *x* and *y* scaling strategies, using a data sharding based approach. This strategy is commonly used to shard databases, but can also be used to scale monolithic applications based on a user characteristic.

This pattern is also useful for premium access policies, where certain users or customers are afforded higher performance guarantees such as high-speed streaming for premium customers on video serving websites. There is a drawback with this approach, which is, it increases the complexity of the application's code. Also, the routing algorithm needs to be continuously tuned as data grows over time. Finding a good candidate for routing is another key factor in this.

The point is the scaling strategies associated with the application architecture. Microservices architecture is symbolic of *y* axis scaling, but for better performance on scale, we have to duplicate or join one or more axis scaling behind the scenes with *y* axis.

Thoughts before scaling

Just creating a microservice is not sufficient. We are doing all this for performance, which, in turn, is for a smooth user experience. We are doing this to increase the users on the platform. So, with an increasing number of users, scaling a microservice becomes important. The more users, the more scaling needed; better scaling gives a better user experience, which gives more users on a platform and moves us back to more scaling. With microservices, everything is more granular, including scalability and managing spikes in demand. No matter how you view the challenges of scalability for microservices, from the customer or end-user perspective, what matters is the performance of the app itself.

Let's take a step back. When it comes to scaling microservices, before doing any dynamic scaling, there are some key points to be thought of:

- **Does infra support it**: You need to know whether the infrastructure on which the system is working supports the dynamic scaling or not. If it supports it, then we can move forward, and if not, then another question is, does it support manual scaling? If it supports manual scaling, then it impacts the rest of the decisions we are going to take about calling policies.

- **Policy for scale up and scale down**: What should the policy be for scaling the system up and down? These policies can be dependent on CPU usage, network traffic in and out, and so on.
- If we have a dynamic instance of our server coming up during peak time, at that time, our monitoring system should be able to monitor that dynamically added instance also.
- Microservice-based architectures are heavily decentralized, because they focus on the successful reusability of individual components, which helps in scaling. Microservices makes it easier to scale because each service can scale independently of other services.
- The increased number of microservices makes containers the most promising option for the deployment of microservices. There are many tools and facilities available to make your microservice containerized. Open source container tools like Kubernetes can be a good choice. Major cloud providers are also adding support for containers to their platform--like Amazon ECS and **Google Container Engine** (**GKE**). They also provide mechanisms for easily scaled container-based applications, depending on load and business rules.
- Containerised deployment also increases the challenge at scale. Monitoring them is one of the issues. Monitoring at a higher scale of container becomes the big data problem. As you have so much data from monitoring, collection of that, making sense of data, taking reactions to that data is a challenge in itself.

There are a few challenges for you to face when scaling containers:

- As we mentioned earlier, there can be many containers, dynamically increasing and decreasing based on load in platform. So, it is crucial to collect logs before these containers come down. This becomes an important point of consideration for scaling in a container.
- Another point of concern is testing. Containerised deployment at scale also magnifies the inherited challenges of container deployment itself. An increased number of containers increases the importance of automated testing, API testing, communication performance, and so on.
- Network security is another issue. At scale, it is going to be another consideration point. How can someone plan or architect a solution where each container has a certain level of networking security, without affecting the performance of the application.

- The most important thing is the team should have enough knowledge of the platform or containers which are going to use the production environment. It's not only the DevOps team; the developer should also be handy with container technology. Developers should also be aware of what is happening inside the container and debugging it.
- The deployment pipeline also increases and becomes more complex. Lots of homework is required on CI and CD. Coming into the microservice world, with so many different services developed independently, the number of deployments increases dramatically.

As per studies, only about 19 percent of developers are using technology for production. Other teams are facing issues such as performance, testing, security, and so on. Also, cloud providers are coming into the field with their container services, solving all these issues. This number is expected to increase.

Practical options for both monitoring and scaling microservices

Monitoring tools in today's world are changing. These tools include reactive monitoring, user behavior monitoring, and so on. If the system finds a failure, then there should be a good enough tool that helps in debugging, such as centralized logging, and so on. In this section, we will visit some of the tools that will be handy in monitoring the microservice. We will now talk about some well-known tools provided by cloud providers such as CloudWatch and so on.

- **QBit** the Java library. It includes a statistics service, which injects a data into StatsD/CodeHale Metrics. This stats service is referred to as statistics engine. This engine provides the data about a microservice and this helps in monitoring that microservice. QBit allows you to write code which can take action on the data provided by the microservice stats engine. StatsD is a network daemon for aggregating statistics. StatsD has many small clients libs for Java, Python, Ruby, Node, and so on.

- **ELK** stands for Elasticsearch, Logstash, and Kibana. Kibana is a visual interface for interacting with large amounts of aggregated log data. Log stash is an open source tool for collecting, parsing, and storing logs for future use. Elasticsearch is a centralized log aggregation and indexing server to index the logs. Kibana's working assumption is that log data analysis is a good way to find problems. Kibana supports multiple log aggregators such as Fluentd, Flume, and so on. The best part of kibana is visualizing query results as graphs and widgets.

- **Dynatrace** is an application performance management tool. It provides monitoring and user experience analysis, which helps developers, testers, and operations ensure their applications work fast and reliably. Dynatrace has patented PurePath Technology for their product, which allows you to trace every single transaction from end to end (browser to database). Dynatrace does lots of things, such as automatic dependency analysis, automatic performance baselining, automatic root cause analysis, and so on. They have autodiscovery for containerized application. They have native support for cloud infrastructure and Docker.

- Sensu does asynchronous checks and handles systems written in Ruby. Some of the features of Sensu are it is basically a server/client architecture (sensu-server, sense-client) on top of RabbitMQ, messaging oriented architecture, great test coverage with continuous integration, designed with modern configuration management systems such as Chef or Puppet in mind, designed for cloud environments, lightweight, and with less than 1200 lines of code. It monitors anything by running a series of checks. Also, Sensu reacts to problems/results by designating handlers. Sensu allows you to reuse monitoring checks and plugins from legacy monitoring tools such as Nagios, Icinga, Zabbix, and many more. It also has a web dashboard providing an overview of the current state of your Sensu infrastructure and the ability to perform actions, such as temporarily silencing alerts.

- **AppDynamics** launched a machine-learning powered APM product in June 2015 to monitor, manage, and analyze complex architectures such as microservices. However, it is more than just an APM. From data collection to processing and then deriving knowledge from your data, AppDynamics provides full visibility into exactly how application performance is affecting your business. AppDynamics shows application performance in real time and automatically discovers application topology and interdependencies. Its APM tool also includes distributed tracing, topology visualization, and dynamic tagging. AppDynamics provides a significant amount of detail to solve performance problems using APM tools and an analytics-driven approach. The agents are extremely intelligent and know when to capture important details and when to simply collect the basics, and this is for every transaction.

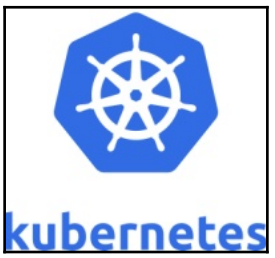
- **Instana,** a monitoring and management platform for web-scale applications, was founded in April 2015. They have an intelligent virtual robot assistant, which they call Stan. Instana uses machine learning and mathematical algorithms to provide dynamic graphs and visualizations. Their intelligent robot assistant helps users monitor and optimize complex applications and architectures via immediate notifications.

- **OpsClarity** has an intelligent monitoring solution combined with in-depth metrics specific to the architectures of web-scale tools. OpsClarity is written in Scala. For UI, it uses Angular JS and nodes at server side. It also makes use of machine-learning libraries, graphing libraries, and large stream processing infrastructure. The technologies supported include data processing frameworks such as Apache Kafka, Apache Storm, and Apache Spark, as well as data stores such as Elasticsearch, Cassandra, and MongoDB.

Apart from these, there are many monitoring tools, such as Prometheus, Netsil, Nagios, New Relic APM, `Foglight`, Compuware APM, and many more. Which tool to use in monitoring entirely depends on the system size, organization need, and financial terms.

- **Scaling** in microservice architecture is not that easy. Frameworks and toolsets like that of Netflix OSS try to automate the process of registering new microservices as much as possible. If containers such as Dockers, Packer, Serf, and so on come into the picture, things become quite complex, but there are some tools available to scale up the container stuff. Here is a list of those tools:

- **Kubernetes** is an open source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure. Kubernetes does not let you worry about what specific machine in your datacenter each application runs on.



- Vamp is an open source, self-hosted platform for managing (micro) service-oriented architectures that rely on container technology. Vamp takes care of route updates, metrics collection, and service discovery, so you can easily orchestrate complex deployment patterns. It is platform agnostic and written in Go, Scala, and React.

- Mesos is an open source cluster manager which manages workload with dynamic resource allocation. The Mesos process runs on all machines, and one machine is defined as a master, which has the responsibility of managing the cluster. All other machines are slaves that work on a particular task using a set number of resources. It schedules CPU and memory resources across the cluster in much the same way the Linux kernel schedules local resources.

- Consul is a nice solution for providing powerful and reliable service-discovery capability to your network and much more. It has multiple components, such as service discovery, health checking, key-value store multi data center. Consul is a distributed, highly available system. It helps in the scaling of a configuration server when you have clusters of services. Every node that provides services to Consul runs a Consul agent. The agent is responsible for health, checking the services on the node, as well as the node itself.

- **AWS container service,** also known as ECS, is a highly scalable high-performance container manager service that supports many containers, such as Docker. It helps scale your own cluster management infrastructure without much headache. The main components involved in container service are EC2 instance, Docker daemon, and ECS agent. ECS manages Docker containers within AWS, allowing users to easily scale up or down, and evaluate and monitor CPU usage.

There are many more tools, such as the following ones:

- etcd
- Deis
- Spotify Helios
- Tutum
- Project Atomic
- Geard
- Docker OpenStack
- Coreos
- Flynn
- Tsuru (as it supports docker-cluster)

There are many tools to help scale, with containers or without containers. Frameworks and tool sets like that of Netflix OSS try to automate the process of registering new microservices as much as possible. The intention of this section was only to make you aware of tools in the respective area of microservice. There are lots of resources available on those tools.

Summary

Monitoring and scaling are two important factors to keep in mind in microservice architecture. They should be thought about at the time of designing microservices. We also understand the reactive monitoring and scaling cube. There are plenty of tools to help an organization to effectively implement monitoring and scaling. We have gone through a number of tools in this chapter. The choice of tool depends entirely on the size of the platform, number of microservices, number of teams and users, or the financial terms of the organization. In the next chapter, we will go through some of the common issues raised in microservice environment and how we can troubleshoot them.

10
Troubleshooting

Migrating to a microservice architecture can go in two directions: first, it might not work at all, or it might not work as expected. The solution would be to roll back to the old monolithic application and land up somewhere in between SOA and the monolithic architecture. Second, the migration activity works perfectly, and it successfully migrated current architecture to a microservice one. In the earlier case, the team will be back to the old historic problem again, which is a known one. In the latter case, the basic old problem might get solved, but you will encounter new sets of problems. In this chapter, we will see what new problems can arise and how we can solve them. This chapter is a discussion of `Chapter 9`, *Monitoring and Scaling*, where we were introduced to new tools. Now, we will use them in troubleshooting.

In this chapter, we will cover the following topics:

- Common issues with microservices
- Techniques to solve or mitigate common issues

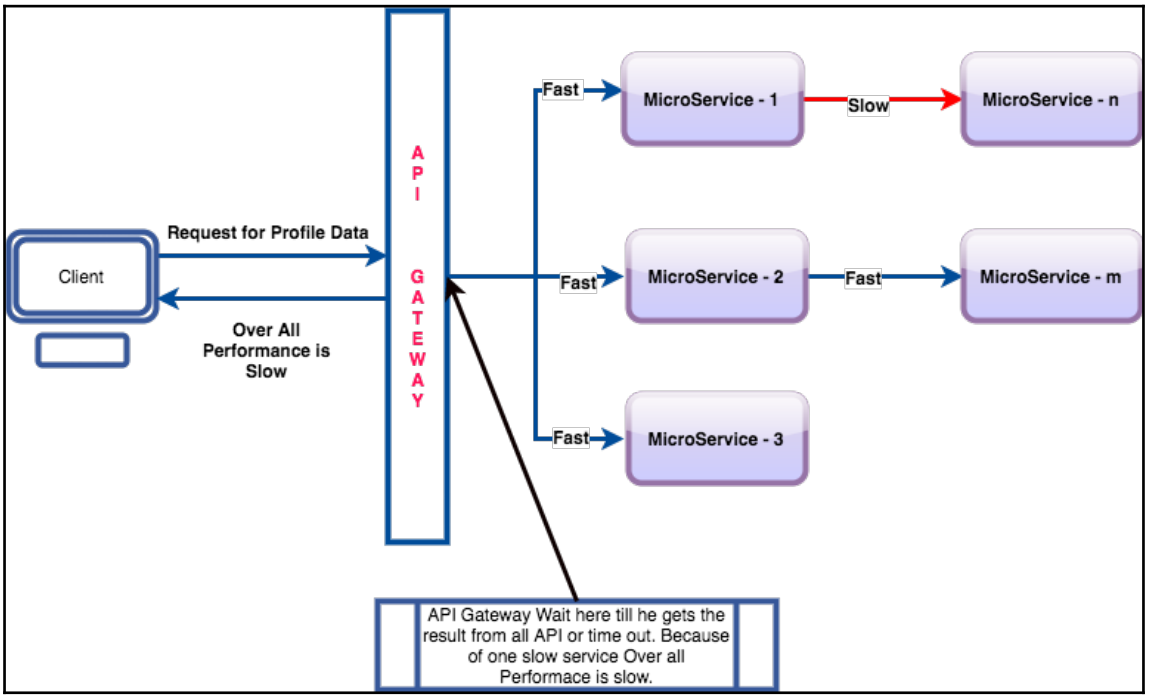
Common issues with microservices

The microservices pattern breaks down the problem statement into smaller components, which have well-defined boundaries. By breaking down the problem into smaller parts, it is easy to handle and there is a better chance of targeting the solution as it is required. Until now, we have seen that all these advantages that come with some trade-offs. These trade-offs come with another set of problems. From scaling to monitoring, everything is multiplied. If you feel monitoring a monolithic architecture was hard, it's 10 times harder with microservices and requires a bigger investment in planning ahead.

In this section, we will understand some of the common issues normally faced by teams while adopting a microservice.

Slow performance

This is the most important and commonly faced issue in the microservice pattern. It is practically impossible to create an application that performs at the same level throughout its life cycle. At some level, the performance of the application takes a hit in microservices, as your code is now distributed into different code components. There could be a scenario where to server one user call, your component has to call five different calls. So, any call can cause a delay, which results in the overall delay of the call. On the other hand, if one of the components is performing slow due to any reason and this component is the connecting point of multiple components, then it will delay the overall performance of the system:

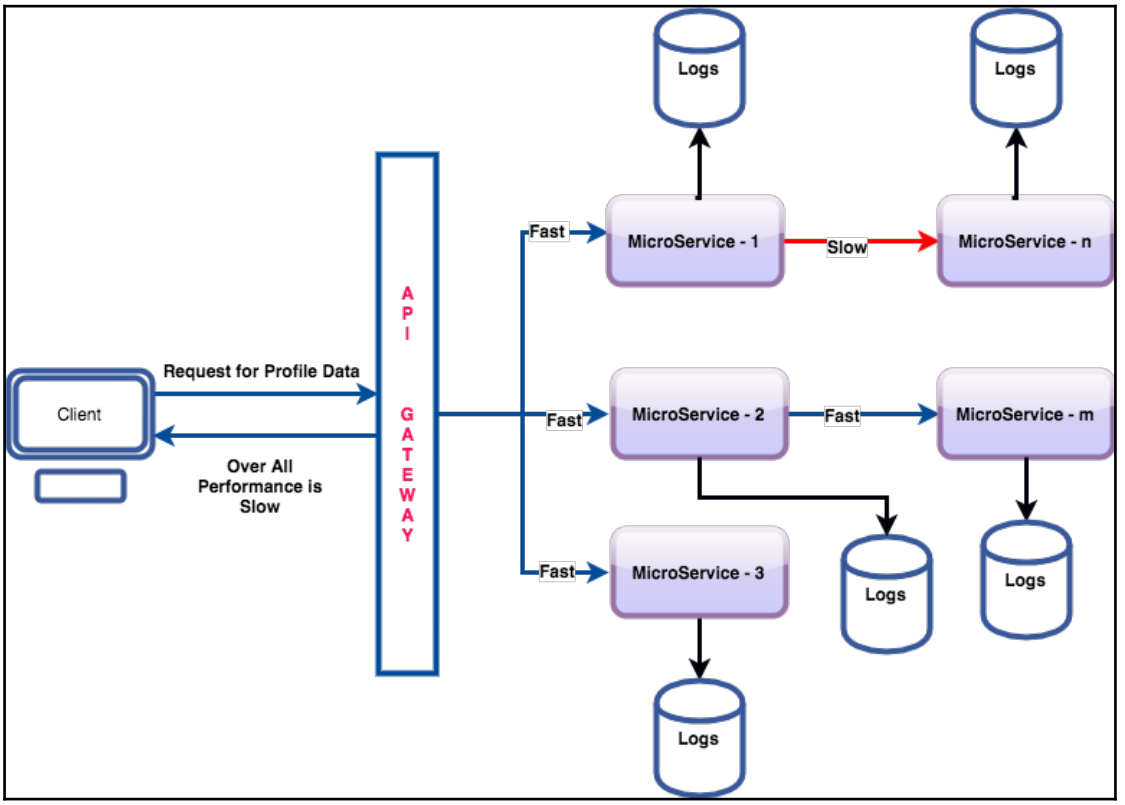


Look at the preceding diagram. Suppose there is a web client that fires the request to get the profile data of a logged-in user. Think of this profile data like a Facebook profile. So, profile data is a mix of posts, reactions, images, friends list, and so on. To get this data in one call, we have used the API Gateway. We described API Gateways in `Chapter 2`, *Defining Microservice Components*. You learned where to minimize the traffic between the client and the server in a single request of the client. The server tried to reason with wall data to the client with the help of the API Gateway. The API Gateway, in turn, is responsible for making internal calls, getting all the data, and sending the final response to the client. During this procedure of gathering all the data, the API Gateway has to fire five different requests to five different microservices. Here, one service is overloaded or responding slowly due to hardware or something like that. This is why the overall performance is degraded. The turnaround time for client requests is increased due to this. So, one bad code can give us slow performance issue, and it is difficult to identify the exact component that is causing the issue.

Performance is not only about speed. It takes so many things into account, such as reliability, failover, and so on, which means we can't blame only the speed. Reliability and failover are also common issues in microservices. If your application is responding pretty quickly but isn't performing reliably, for example, sometimes it will give you result with 200 response code but sometimes it gives a 404 error, this is also a performance issue. So, this is speed at the cost of reliability and reliability at the cost of speed. User experience will be affected in any of the cases mentioned earlier. There should be a fine balance between speed and reliability.

Different logging locations from different programming languages

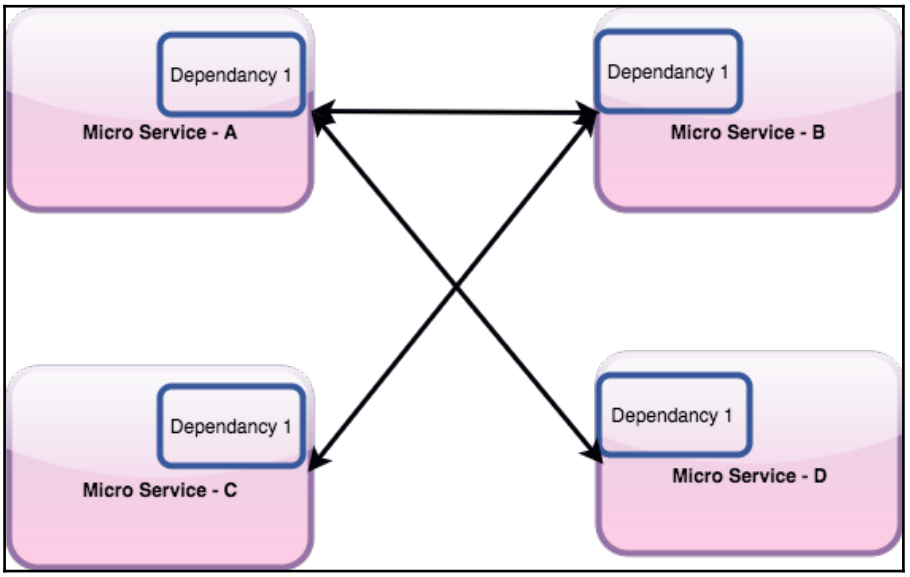
As in microservices, we have different services running in different locations. They are isolated from each other, so their logs are also at different locations. Also, there are chances that the format of logs is quite different. While debugging any particular traffic call, the developer has to look into many different log files, which have other data as well. As in our last example, microservice 1, 2, 3 , n, and m all have different log files, different locations, or different servers. It is not an easy task for any developer to identify which particular component is behaving slowly.



If you increase the complexity here, for instance, rather than 5 microservices, one call deals with 10 microservices, there will also be a lot of traffic on the web site, depending upon the log level, the log file also become in gigabytes. This will make it very hard to identify the exact issue.

Coupling or dependency issues in multiple components

Coupling or dependency means the code of one service or library has an effect on the other. However as per microservice architecture, each service should be independent and not have an impact on other services with well-defined boundaries. In spite of that, for common code collection, developers develop a library kind of code, which they will insert in every microservice as a helper. A common example is if one component is serializing some data as an API call response, which, at the client end, should be deserialized as the same object. This could be a case where the same model is needed in four different components. As practice, developers made some library code and used it in all components. This, in turn, creates two issues: one is, if you have a library to do many of the common things, there are higher chances that a new microservice will be developed in the same language in which you have the library code. This will impact the freedom of using any language in microservice. They are all used to bind the same programming language:



Another scenario is identified as: what if you have the same dependency in two or more services. If that dependency is updated, then, in what sequence should you update the services having the same dependency? All of them are interacting with each other; updating in one can cause some failed calls in another component.

Per-day deployment for more number of services

Microservices are all about breaking things down to individual components. Having so many components results in many repositories and thus into many deployment cycles. Also, we are talking about speed, which increases the number of frequent deployment on a daily basis. As a ripple effect of this, infrastructure operation work may increase such as they have to maintain the deployment speed, to create separate deployment pipeline for each and every microservice, making sure that one microservice is not affecting another microservice during deployment and so on. If you have services more than 20 that talk to a new microservice, testing all these microservices would take of a lot of time. This concludes to us the solution that we should have automatic testing, which, in turn, starts impacting the resources. For example, then, there should be one another environment for testing. More servers means more money. Running all these microservices also uses quite a large number of servers than a monolithic application. As the number of environment/servers increases in an organization, DevOps dependency or involvement also starts increasing. So, high DevOps costs also a common issue here.

Monitor many services for performance degradation or issues

As mentioned in the previous section, if you are deploying more service per day, then monitoring becomes another important impact. Sometimes, we are not able to maintain all kind of data of monitoring and make sense of it. You may have a dashboard to show various parameters of servers and their processes such as CPU usage, disk space, traffic bytes in and out, and so on. Having so much information and making sense of it and reacting to it is not always easy. Newly deployed services may be out of network reachability of another machine. So, deployment works fine, but these components are not talking to other components. Monitoring can show us that if service and server is reachable to monitor service, it is hard to monitor whether every service is reachable to every other desired service or not. Monitoring is also breaking down per service and losing their power for the system as a whole. Because of so many components, the challenge here is to take control back to centralize using proper tooling.

Relation between logs and different components

As, in microservices, there are many components each can be running on the same or different instance. As we have mentioned earlier, also, there is no good view for over ll picture of logs in microservices Architecture. So it is difficult to understand what is happening to any particular request. If something goes wrong, someone has to be debug, and it is not necessary that the same person needs to have knowledge of all technology used in platform and also have knowledge of what all microservices are doing. The only thing they can come to is check the logs and make sense of them to get the overall picture of what is happening in the flow. Then, they encounter another problem. Even if they have all the logs in the same place (after some effort), they can't pull out information from GBs of data. As different components are developed in different languages and by different developers, without having any good standards, there are high chances that the format of logs does not match between different components. For instance, we can see a Java sample log format here:

Java log file sample output:

```
2017-05-26 15:34:53.787 INFO 13262 --- [io-10000-exec-2] i.s.c.controller.
< Controller Class> : <Log content>
```

It has date, time, log level, class name with package, and log content. Python standard logging is different, same as in Ruby, so in short, different languages have different logging standards and different logging libraries as well. It is not a good change to use any Regex-capable tool to find something between logs.

Techniques to solve the common issue

In the previous section, we looked at some common issues we face in microservice architecture. We will try to find out a solution pattern for them in this section.

Steps to help with performance issues

Performance in microservice does not only depend on the code. It has to include issues such as network latency, message serialization/deserialization, dynamic loads within the application, and so on. This actually increases the scope of testing in terms of performance. The first step would be to increase the resources. Scale your application vertically, increase the server resources that will result in some performance gain, but adding more resources to the server cannot always help you. Adding load balance is another approach. You can add load balance in front and adding more server in backend to handle the scale. This should even be optimized when traffic is high and there is an increase in the number of servers. One observes the patterns with old data and can plan accordingly to create some automation, which will predict the traffic and increase the number of servers to handle traffic.

Any server that is not responding to requests at all or responding very slow, like taking 5 seconds to respond to one request, while other servers are responding to it in milliseconds should be removed from serving. Load-balancing service can have multiple techniques. As per data available on the Internet, round-robin techniques in load balancing gives the 95 percent success rate, while another more smart technique the least loaded server will give you a 99 percent success rate.

Besides the heavy traffic issue, there could be an issue inside the compose code also. As a client API call can make many services talk to multiple services, it will be hard to point the source of performance problem of slow response time. There are tools that give you performance measurement. For example, if you have a web server such as Nginx or Apache in front of your application, which logs the response time, you can write an alert on that column. If it increase from some threshold, then add a new server and remove the slow responding server from the load balancer. Another solution is the user should monitor the performance of each individual microservice. Graphite, Statsd, or cloud-based services such as New Relic provide rich, visual dashboards, which we discussed in the previous chapter, where problems can be discovered.

Handle logging in different locations and from different languages

Debugging always starts from logging. Due to a large number of microservices running in the system, if any problems occur in the system, the developer wants to dig in. However, as there are so many log files of different components, finding one call trace from this big data is not easy is rather time consuming. The solution to this problem is to use log-aggregation tools such as log stash with Kibana. Both of them are open source tools. Another example is Loggly, which is cloud-based log aggregator tool. Logstash collects the log file in one place, and Kibana allows you to search inside the logs. Both of the mentioned tools are explained in `Chapter 9`, *Monitoring and Scaling*. The problem is not always in the component; it can be in the supporting tool. Like Java service running in JVM, but someone has given very low RAM to JVM, which, in turn, slows down the processing of your component as JVM is frequently busy garbage collecting.

Another problem to handle is log format. All components should agree to one format. Normally, in these cases, all components start emitting the logs in a well-defined JSON format, which makes querying on the logs easy. It is more like storing data in Elasticsearch as JSON and querying over it, which makes it pretty fast. Loggly, a cloud-based log aggregator, supports the JSON-based logged. It also has a nice interface to show logs by tag or in a particular time frame, and so on.

Even though we have aggregated the logs, tracing a client call inside the platform is still an issue. If one service issues an API call to the next component, and that leads to another one, and so, on then it is very hard to understand what happens to a particular call in the next component. How can a developer figure out about one particular request and what happens to it? Flow ID/Composition ID/Correlation ID, whatever you may call it, this unique ID attaches to each call platform in the first place and that gets passed across calls, which allows you to track each request. This ID is attached per request initiated. This requires some effort in coding. One has to generate it and log it every time, and then pass it to another microservice; but it is a trade-off. This gives an overall picture of what is happening to a particular request. For any debugging, this can be a very good starting point.

Many commercial software packages, for example, Microsoft SharePoint, are using this pattern to help with problem solving.

Dependencies between services

While defining this problem, we mentioned the common library lying in a different microservice, which will bound us to use any one particular language. To avoid this, we should adopt another bad habit, which is code redundancy. To make microservice completely isolated from each other, code replication is used. This will solve the issue of dependency and will give the freedom of language choice also. This is a tradeoff, but it is not the best solution. After some time, when microservice architecture evolves more, then we might have more practices in action, and that will give us a better idea. Also, whenever we change any dependency, we should not break the backward compatibility until all the services come to a new version. When all the services come to the latest version, then we can remove the old version.

High DevOps involvement

With so many environments in place, so many servers running in the platform, there is a need for DevOps guys to be available 24x7. If someone wants to release a service, they require a DevOps guys to build a release and deploy it. In monolithic, any release is a big thing, happening perhaps months. Any failure will bring the whole platform down and increase the downtime. So, DevOps guys are always required. There are two ways to handle this issue.

Use smart tools

With the use of smart tools such as Jenkins, Travis, Chef, and Puppet, with the inclusion of some custom scripts, you should completely automate deployments of microservices. One-button deployment should be implemented and deployed.

Use smart developers

Make your developer learn the DevOps work also. Developers are not supposed to understand the whole work of DevOps, but while migrating to a microservice architecture , a developer should come with the mindset of ownership of their microservice, and their ownership is not limited only to coding. They should be responsible for knowing each step to make their microservice run in a production environment.

Using a combination of these two methods, DevOps work can be distributed to some extent. Reactive monitoring also helps DevOps. We will discuss this in the next chapter. Deploying in cloud rather than on a standalone server will help operations work a lot. Elasticity is the best part provided by cloud. It helps the server handle a sudden spike in traffic. One can make different rules in it. For example, if the response time of any service increases to 400ms, deploy several more servers.

Monitoring

With such a large system broken into pieces and with work distributed, it is very hard to keep eye on each piece to check whether it is working properly. The first step would be to bring the whole monitoring system to a centralized place. No matter how many components you have running separately, you should bring logging and monitoring to a centralized place to debug better and faster.

The next step would be reactive monitoring or synthetic monitoring. This type of monitoring should be done inside the microservice and among microservices. If one of the microservices went down or some service stopped talking to databases, or one service is facing an issue with disk space and so on, it is very important for us to know it before our live audience notices it. That's why, we need reactive monitoring. If our monitoring is not able to fix the issue, at least it should notify us about it as soon as possible. There are many tools that can help us achieve active monitoring, which are explained in the last chapter. Other than those, there is one more tool called zip kin, which is like a debugger for distributed app.

Zipkin is an open source tool developer by Twitter to trace and debug into microservice. This is inspired by Google's Dapper Idea. Zipkin depends on Apache ZooKeeper. Zipkin aggregates timing data that can be used to track down latency issues. This tool automatically assigns the unique identifier for each request. For Zipkin to work, each microservice is instrumented with a Zipkin library that the service then uses to identify the request's entry points.

Summary

In this chapter, we have seen some common issue into microservice and how we should deal with those. Most of the issues can be solved using proper tooling, which we have explained in the last chapter. Rest of the issues can be solved by proper practicing. I hope this book is able to give you some insight into using microservice in the practical world.

Index

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z