Spring Microservices IN ACTION

John Carnell

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Spring Microservices in Action
Version 1

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Spring Microservices in Action*. It is an exciting time to be a Spring developer, because the Spring Framework has done something that very few frameworks have been able to do: stay relevant in the face of constant technical change. As more and more applications have moved to the cloud, the Spring development community has responded with Spring Boot and Spring Cloud. These two Spring frameworks will allow you to quickly build microservices that are ready to be deployed to a private corporate cloud or a public cloud like Amazon Web Services (AWS) or Pivotal's CloudFoundry.

You do not have to an in-depth cloud experience to benefit from this book. To take advantage of this book you should have an established background in Java and Spring (2-3 years). Some building web-based services (SOAP or REST) is helpful. Other then that, you just need a willingness to learn.

I first encountered Spring Boot and Spring Cloud while working as an Integration Architect at a Fortune 500 financial services company. More and more of my job did not involve building applications, but rather building services that were going to be deployed to a cloud. I found existing SOAP-based web services, even Spring-based ones, were tedious and time-consuming to deliver. The SOAP protocols were complex and difficult to work with. Furthermore, as we began deploying more and more cloud-based services, the operational infrastructure needed to keep a services-based application running was immature. I spent many an evening as part of our critical situation team trying to bring back up an application that had crashed because of a failure in one or more of the services it depended on.

The concept of a microservice architecture is appealing because it shifts development away from building heavy services based on the SOAP protocols to a much simpler REST/JSON-based service model where an individual service is very constrained in the work is does. However, any architecture is only as good as the tools and languages that can implement it. When I discovered the Spring Boot and Spring Cloud frameworks, I realized that these tools provided the capabilities I needed to build microservice architectures, while still leveraging my years of experience with Java and Spring.

I was so impressed with the power these frameworks brought, that it one of the reasons why I chose to write this book. Books are a labor of love and one the deepest expressions of an authors experiences and knowledge. I hope you enjoy this book. While it has been a lot hard work to get to this point in the MEAP, I look forward to hearing your comments and feedback as you explore this existing technologies with me.

—John Carnell

brief contents

1

Welcome to the cloud, Spring

This chapter covers

- What are microservices
- The drivers for changing to microservices
- The cloud and how microservices are a great natural fit for cloud applications
- Different patterns of microservice development
- How Spring Boot/Cloud supports microservice developments

The one constant in the field of software development is that we as software developers sit in the middle of a sea of chaos and change. We all feel the churn as new technologies and approaches appear suddenly on the scene that cause us to re-evaluate how we build and deliver solutions forour customers. One example of this churn is the rapid adoption by many organizations of adopting and building applications using microservices.

This book is going to introduce you to the microservice pattern and why you should consider building your applications with them. We are also going to look at how we can build microservices using Java and two Spring framework projects: Spring Boot and Spring Cloud. If you are a Java developer, Spring Boot and Spring Cloud are going to provide an easy migration path from building traditional monolithic Spring applications to microservice applications that can be deployed to the cloud.

The concept of a microservice originally crept into the software development community's consciousness around 2014. A microservice is ansoftware design approachbased around the concept of building applications out of very small, loosely coupled and distributed services.

- Application logic is broken down into very small grained components with well defined boundaries of responsibility that coordinate together to deliver a solution.
- Each component has a very small domain of responsibility and is deployed completely

independently of one another. Microservices should have responsibility for a single part of a business domain. Also a microservice should be reusable across multiple applications.

- Microservices communicate based on a few basic principles (notice I said principles, not standards) and employ a very lightweight communication protocols (e.g. HTTP and JSON) for exchanging data between the service consumer and service provider.
- The underlying technical implementation of the service are irrelevant because the applications always communicate with a technology neutral protocol (JSON is the common). This means an application built using a microservice application could be built with multiple languages and technologies.
- Microservices by their small, independent and distributed nature allow organizations to have small development teams with well-defined areas of responsibility. These teams might work to towards a single goal of delivering an application, but each team is responsible for only the services they ae working.

I often joke with my colleagues that microservices are the gateway drug for building cloud applications. You start building microservices because they give you a high degree of flexibility and autonomy with your development teams, but you and your team quickly find that the small, independent nature of microservices makes them easily deployed to the cloud. Once the services are in the cloud, their small size makes its easy to start up large instances of the same service and suddenly your applications become more scalable and with some forethought more resilient.

1.1 What is Spring and why it's relevant to microservices

Spring has become the de-facto development framework for building Java-based applications. At its core Spring is based on the concept of dependency injection. In a dependency injection framework that allows you to more easily manage large Java projects by externalizing the relationship between objects within your application through convention (and annotations) rather then have those objects have hard-coded to know about it each other.

Spring's rapid inclusion of features drove its utility and the framework quickly became a lighter weight alternative for enterprise application Java developers looking for an alternative to building applications using the J2EE stack. The J2EE stack, while powerful was considered by many bloat with many features that were never used by the application development teams. Further, a J2EE application forced you to use a full-blown (and heavy) Java application server to deploy your applications.

What is amazing about the Spring framework and a testament to its development community is is ability to stay relevant and re-invent itself. The Spring development team quickly saw that many development teams were moving away from monolithic applications where the applications presentation, business and data access logic were packaged together and deployed as a single artifact. Instead teams were moving to highly distributed model where services were being built as small, distributed services that could be easily deployed to

the cloud. In response to this shift, the Spring development team launched two projects Spring Boot and Spring Cloud.

Spring Boot is a re-envisioning of the Spring framework. While it embraces core features of Spring, Spring Boot strips away many of "enterprise" features found in Spring and instead delivers a framework built geared towards Java-based, REST (Representational State Transfer)[1] orientated microservices quickly. With a few simple annotations, a Java developer can build a REST microservice that can be package and deployed without the need for an external application container.

> **NOTE** While we will cover REST in more detail in chapter 2, the core concept behind REST is that your services are invoked via the HTTP protocol, embrace the use of the HTTP verbs (GET, POST, PUT and DELETE) to represent the core actions of the service and use a very lightweight web orientated data serialization protocol, like JSON for requesting and receiving data from the service.

Since microservices have become one of the more common architectural patterns for building cloud-based applications, the Spring development community has given us Spring Cloud. The Spring Cloud framework makes operationalizing and deploying microservices to a private or public cloud simple. Microservice applications are highly distributed with many moving components. Spring Cloud wrappers several popular cloud management microservice frameworks under a common framework and makes the use and deployment of these technologies as easy to use as just annotating your code. We will cover the different components within Spring Cloud later on in the chapter.

1.2 What you will learn in this book

This book is about building microservice based applications using Spring Boot and Spring Cloud that can be deployed to a private cloud run by your company or a public cloud like Amazon, Google or Pivotal.With this book, we will cover with hands-on examples:

- What is a microservice and what are the design considerations that go into building a microservice-based application
- When shouldn't you build a microservice-based application.
- How to build microservices using the Spring Boot
- What are the core operational patterns that need to be in place to support microservice application, particularly a cloud-based application
- How we can use Spring Cloud to implement these operational patterns
- How to take take what we have learned and build a deployment pipeline that can be used to deploy to multiple cloud providers

---

[1] While we cover REST later on in Chapter 2, it always worthwhile to read Roy Fielding's PHD dissertation on building REST-based applications (http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) is still one of the explanations of REST available.

1.3 Why is this book relevant to you

If you have gotten this far into reading chapter 1, I suspect that:

- You are a Java developer.
- You have some kind of background in Spring.
- You are interested in learning how to build microservice-based applications.
- You are interested in how you leverage microservices for building cloud-based applications.
- You want to know if Java and Spring are relevant technologies for building microservice-based applications.
- You are interested to see what goes into deploying a microservice-based application to the cloud.

I chose to write this book, because while I have seen many good books on the conceptual aspects of microservices, I could not a find a good Java-based book on implementing microservices. While, I have always considered myself a programming language polyglot (knows and speaks several languages), Java is my core development language and Spring has been the development framework I "reach" for whenever I build a new application. When I first came across Spring Boot and Spring Cloud, I was blown away. Spring Boot and Spring Cloud greatly simplified my development life when it came to building microservice based applications running in the cloud.

Enough talk. I have talked a lot this point with very little code examples. Let's shift gears for a moment and walkthrough building a simple microservice using Spring Boot.

1.4 Building a microservice with Spring Boot

I have always been of the opinion that a software development framework is well thought out and easy to use if it passes what I affectionately call the "Carnell Monkey Test." If a monkey like me (the author) can figure out a framework in 10 minutes or less it has promise. That is how I felt the first time I wrote a sample Spring Boot service. I want you to have to the same experience and joy. So let's take a minute to write a simple "Hello World" REST-service using Spring Boot.

Figure 1.1 shows what our service is going to do and the general flow of how Spring Boot microservice will process a user's request.

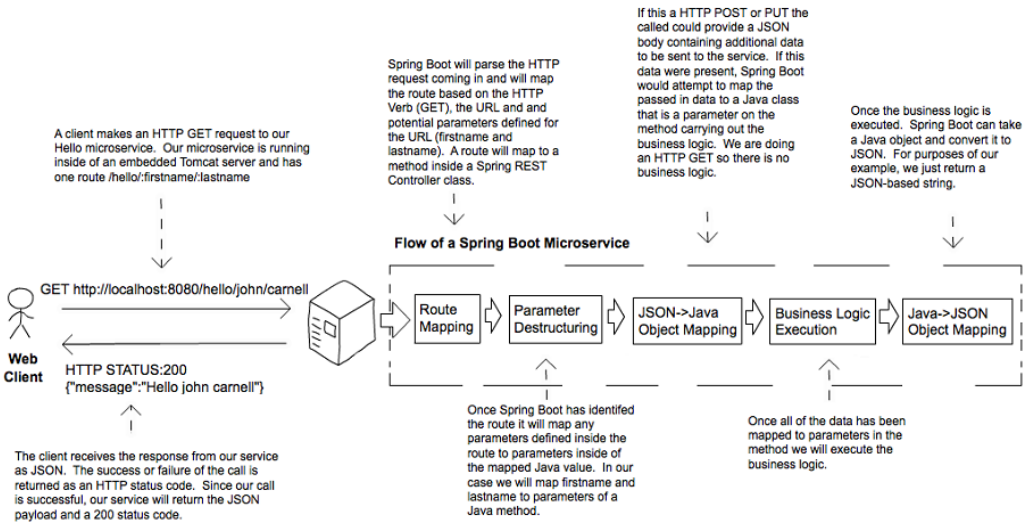

Figure 1.1 A user calling our Hello World microservice

This example is by no means exhaustive or even illustrative of how you should build a production-level microservice service, but it should cause you to take a pause. We are not going to go through how to setup the project build files or the details of the code until Chapter2. If you would like to see the maven pom.xml file and the actual code, you can find in the Chapter 1 section of the downloadable code.

So for our example we are going to have a single Java class call `src/com/thoughtmechanix/application/simple/Application.java` that will be used to expose a REST endpoint called `/hello`.

Listing 1.1 shows the code for our `Application.java`.

Listing 1.1 Hello World with Spring Boot: A (very) simple Spring Microservice

```
package com.thoughtmechanix.simpleservice;

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

@SpringBootApplication                  (1)
@RestController                         (2)
@RequestMapping(value="hello")          (3)
public class Application {

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

```
    }

    @RequestMapping(value="/{firstName}/{lastName}",    ❹
                            method = RequestMethod.GET)
    public String hello( @PathVariable("firstName") String firstName,  ❺
                         @PathVariable("lastName") String lastName) {
        return String.format("{\"message\":\"Hello %s %s\"}",    ❻
                              firstName, lastName);
    }
}
```

❶ Tells the Spring Boot framework that this class is the entrypoint for the Spring Boot framework.
❷ Tells Spring Boot we are going to expose the code in this class as a Spring RestController class.
❸ All URLs exposed in this application will be prefaced with /hello prefix.
❹ Spring Boot will expose an endpoint a GET-based REST endpoint that will take two parameters in it firstName and lastName.
❺ Maps the firstName and lastName parameters passed in on the URL to two variables passed into the hello function.
❻ Returns a simple JSON string.

In listing 1.1 we are basically exposing a single GET http endpoint that will take two parameters (`firstName` and `lastName`) on the URL and then return a simple a JSON string that has a JSON payload containing the message "Hello *firstName lastName*". So if we were to call the endpoint `/hello/john/carnell` on our service(which will be shown shortly) the return of the call would be:

```
{"message":"Hello john carnell"}
```

So lets actually fire up our service. To do this we are going to go to the command prompt and issue the following command from the command-line:

```
mvn spring-boot:run
```

This command, mvn, will use a Spring Boot plug-in to start the application using an embedded tomcat server.

Java vs. Groovy and Maven vs. Gradle

The Spring Boot framework has strong support for both Java and the Groovy programming language. You can build microservices with Groovy and and no project setup. Spring Boot also supports both maven and the gradle build tools. I have chosen to limit the examples in this book to Java and Maven. As a long-time Groovy and Gradle aficionado, the author has a healthy respect for the language and the build tool, but in an effort to keep the book manageable and the material focus, I have chosen to go with Java and Maven to reach the largest audience.

If everything starts correctly, you should see the following from your command-line window:

Figure 1.2 Our Spring Boot Service will communicate the endpoints exposed and the port of the service

If you look closes at the screen shot above in Figure 1.1 you will notice two things. First, a tomcat server was started on port 8080. Second, a GET endpoint of `/hello/{firstName}/{lastName}` is exposed on the server.

So let's call our service using a command-line tool called curl and see what is returned. The curl command to issue is:

```
curl http://localhost:8080/hello/john/carnell
```

When the command above is issued, we should see the following response.

```
jc200941ml2:V2 johncarnell1$ curl http://localhost:8080/hello/john/carnell
{"message":"Hello john carnell"}jc200941ml2:V2 johncarnell1$
```

Figure 1.3 Response from the /hello endpoint

Obviously this simple example does not demonstrate the full power of Spring Boot. But what it should show is that you can write a full HTTP JSON REST-based service with route mapping of URL and parameters in Java with as little as 25 lines of code. As any experienced Java developer will tell you, writing anything meaningful in 25 lines of code in Java is extremely difficult. Java, while being a very powerful language, has acquired a reputation of being wordy compared to other language.[2]

We are done with our brief tour of Spring Boot and will cover it in more detail in Chapter 2. However, just because we can write our applications using a microservice approach, does this mean we should? In the next section we are going to walkthrough why and when a microservice approach is justified for building your applications.

[2] Shhh don't tell anyone, but this Java programmer has a habit of also writing Clojure code. That's a book for another day

1.5 Why change the way we build applications?

We are at an inflection point in history. Almost all aspects of modern society are now wired together via the Internet. Companies that used to serve local markets are suddenly finding that they can reach out to a global customer base. However, with a larger global customer base has also come global competition. These competitive pressures mean the following forces are impacting the way developers have to think about building applications:

- **Complexity has gone way up.** Customers expect that all parts of an organization know who they are. "Siloed" applications that talk to a single database are no longer the norm. Today's applications need to talk to multiple services and database residing not only inside a company's data center, but also to external service providers over the Internet.
- **Customers want faster delivery.** Customers no longer want to wait annually for the next release or version of a software package. Instead, they expectthe features in a software product to be unbundled so that new functionality can be released without having to wait for an entire product release.
- **Performance and scalability**. Global applications make it extremely difficult to predict how much transaction volume is going to be handled by an application and when that transaction volume is going to hit. Applications need to be able to scale up across multiple servers quickly and then when the volume needs have passed, scale back down.
- **Customers expect their applications to be available.** Since, customers are one click away from a competitor, a company's applications must be highly resilient. Failures or problems in one part of the application should not bring down the entire application.

To meet the expectations listed above, we as application developers have to embrace the paradox that to build high-scalable and highly redundant applications we need to break our applications into very small services that can be built and deployed independently of one another. If we "unbundle" our applications and move them away from a single monolithic artifact we can build systems that are:

- **Flexible.** Decoupled services can be composed and rearranged to quickly deliver new functionality. The smaller the unit of code that one is working with the less complicated it is to change the code and the less time it takes to test deploy the code.
- **Resilient.** Decoupled servicesmeans an application is no longer a single "ball of mud" where a degradation in one part of the application causes the whole application to fail. Failures can be localized to a small part of the application and contained before the entire application experiences an outage. This also enables the applications to degrade gracefully in case of un-recoverable error.
- **Scalable**. Decoupled servicescan easily be distributed horizontally across multiple servers enabling the ability to scale the features / services appropriately. With a

monolithic application where all of the logic for the application is intertwined the entire application needs to scale even if only a small part of the application is the bottleneck. Scaling on small services are localized and much more cost-effective.

To this end as we begin our discussion of microservices keep the following in mind:

SMALL, SIMPLE AND DECOUPLED SERVICES

=

SCALABLE, RESILIENT AND FLEXIBLE APPLICATIONS

1.6 What exactly is the cloud

The term "cloud" has become overused. Every software vendor has a cloud and everyone's platform is cloud-enabled, but if you cut through the hype there are really three basic models of cloud-based computing. These are:

- Infrastructure as a Service
- Platform as a Service
- Software as a Service

To better understand these concepts let map the everyday task of making a meal to how a different models of cloud computing. When we want to eat a meal we have four choices:

1. We can make the meal at home.
2. We can go to the grocery store and buy a meal pre-made that we have to heat up and server.
3. We can a meal delivered to our houses
4. We can get in the car and eat at restaurant.

Figure 1.4 shows each model:

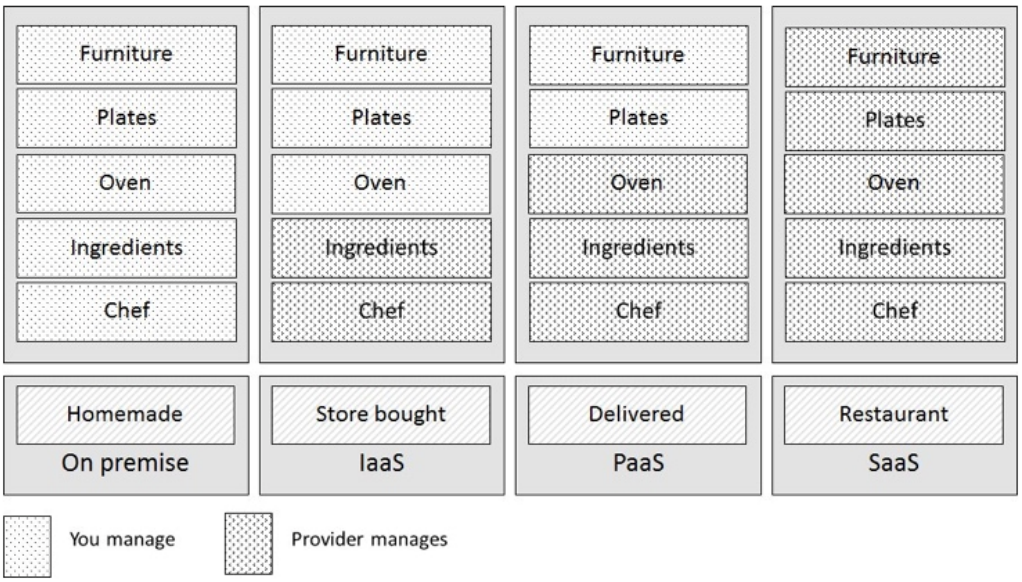

Figure 1.4 The three core models of cloud computing

The difference between these options is really about whose responsible for cooking these meals and where the meal is going to be cooked. In the on-premise model, eating a meal at home requires you to do all of the work using your own oven and ingredients already in the home. A store-bought meal is like using the Infrastructure as a Service (IaaS) model of computing. We are using the store's chef and oven to pre-bake the meal, but we still are still responsible for heating the meal and eating it at house (and cleaning up the dishes afterwards).

In a Platform as a Service (PaaS) model we still have some responsibilities for the meal (e.g. we have to supply the plates and furniture), but we further rely on a vendor to take care of the core tasks associated with making a meal. In the Software as a Service (SaaS) model, we go to a restaurant where all the food is prepared for us. We eat at the restaurant and then we pay for the meal when we are done. We also have no dishes to prepare.

The key items at play in each of these models is one of control: who is responsible for maintaining the infrastructure and what are the technology choices available for building an application.

1.7 Why the cloud and microservices

One of the core concepts of a microservice-based architecture is that each service is packaged and deployed as is its own discrete and independent artifact. Services instance should be

able to brought up quickly and each instance of the service should be indistinguishable from each other.

As a developer writing a microservice sooner or later you are going to have to decide whether your service is going to be deployed to a:

- **Physical Server.** While you can build and deploy your microservices to a physical machines few organizations do this because physical servers are constrained. You can not quickly ramp up the capacity of a physical server and scaling horizontally your microservice across multiple physical servers can become extremely costly.
- **Virtual Machine Images.** One of the key benefits of microservices is being able to quickly startup and shutdown microservice instances in response to scalability and service failure events. Virtual machines are the heart and soul of the major cloud providers. A microservice can be packaged up in a virtual machine image. Multiple instances of the service can then be quickly deployed and started in either a IaaS private or public cloud.
- **Virtual Container**. Virtual container are a natural extension to deploying your microservices on a virtual machine image. Rather then deploying your service to a full virtual machine, many developers will deploy their services as Docker (or equivalent container technology) containers to the cloud. Virtual containers run inside of a virtual machine and allow you to segregate a single virtual machine into a series of self-contained processes sharing the same virtual machine image.

For this book, all the microservices and corresponding service infrastructure will be deployed to an IaaS-based cloud provider. This is the most common deployment topology used for microservices:

- **Simplified Infrastructure Management.** IaaS cloud-providers give you the ability to have the most control over your services. New services can be started and stopped with simple API calls. With an IaaS cloud solution, you only pay for the infrastructure that you use.
- **Massive horizontal scalability.** IaaS cloud providers allow you to quickly and succinctly start service (and their corresponding services). This capability means you can scale services quickly and also allows you to quickly route around misbehaving or failing servers.
- **High redundancy through geographic distribution.** By necessity IaaS providers will have multiple data centers. By deploying your microservices using an IaaS cloud provider you can gain a higher level of redundancy beyond just using cluster in a data center.

Whynot PaaS-based Microservices?

Earlier in the chapter we identified there are three types of cloud platforms (Infrastructure as a Service, Platform as a Service and Software as a Services). For this book we have chosen to focus specifically on building microservices using an IaaS-based approach. While some cloud providers will let you abstract away the deployment infrastructure for your microservice.

For instance, Amazon, CloudFoundry and Heroku give you the ability to deploy your services without having to know about the underlying application container. They provide a web interface and APIs to allow you to deploy your application as a WAR or JAR file. The setup and tuning of the application server and the corresponding Java container are abstracted away from you. While this is very convenient, each cloud provider's platform have different idiosyncracies related to their individual PaaS solution.

An IaaS approach, while more work, is portable across multiple cloud providers and also allows us to reach a wider audience with our material. Personally, I have found that PaaS based cloud solutions can allow you to quickly jump start your development effort, but once your application reaches enough microservices, you starting needing the flexibility the IaaS style of cloud development provides.

The services built in this book will be packaged as Docker containers. One of the reasons why I chose Docker is that as a container technology, Docker is deployable to all of the major cloud providers. Later in Chapters 8 and 9, I will be demonstrating how to package our microservices using Docker and then deploy these containers to Pivotal's and Amazon's cloud platforms. (CloudFoundry and Amazon Web Services respectively)

1.8 Microservices are more then just writing the code

While the concepts around building individual microservices are easy to understand, actually running and supporting a robust microservice application (especially when running in the cloud) involves a lot more then writing the services code. It involves having to think about how your services are going to be:

- **Right-sized.** How do we ensure that our microservices are properly sized so that we do not have a microservice take on too much responsibility? Remember, properly size a service allows us to quickly make changes to an application and reduces the overall risk of an outage to the entire application?
- **Manageable.**How do we manage the physical details of service invocation when in a microservice application you might have multiple service instances can quickly start and shutdown?
- **Resilient.** How do protect our microservice consumers and the overall integrity of our application by routing around failing services and ensuring that we take a "fail-fast."
- **Repeatable.** How do we ensure that every new instance of our service brought up is guaranteed to have the same configuration and code base as all the other service instances in production?
- **Scalable.** How do we leverage asynchronous processing and events to minimize the direct dependencies between our services and ensure that we can gracefully scale our

microservices?

This book is going to take a patterns based approach as we answer the questions above. Specifically, we are going to cover the following four categories of patterns:

- Core Microservice Development Patterns
- Microservice Routing Patterns
- Microservice Client Resiliency Patterns
- Microservice Build/Deployment Patterns

Let's walk through these patterns in more detail.

1.8.1 Core Microservice Development Pattern

The core development patterns really deals with the basics of building a microservice. Figure 1.5 highlights the topics we will cover around basic service design.

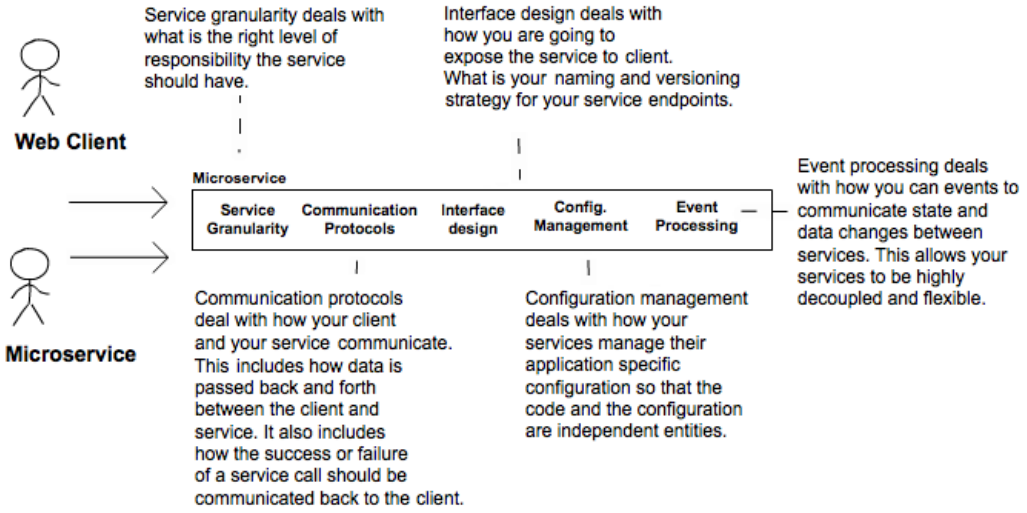

Figure 1.5 – In building a simple service we have to consider these topics

Let's walk through these topics in a little more detail.

- **Service Granularity.** How do you approach decomposing a business domain down into microservices so that the microservice has the right level of responsibility? Making a service too coarse-grained with responsibilities that overlap into different business problems domains makes service difficult to maintain and change over time. Making the service too fine-grained increases the overall complexity of the application and really just turns the service into a data later. Service granularity will be covered in chapter 2.

- **Communication Protocols.** How will developers communicate with your service? We will go into why JSON is the ideal chose for microservices and has become the most common choice for sending and receiving data to microservice. The topic of communication protocols will be covered in Chapter 2 of this book.
- **Interface Design.** What is the best way to design the actual service interfaces that developers are going to use to call your service? How do you structure your service URLs to communicate service intent? What about versioning your services? A well design microservice interface makes using your service intuitive. Interface design will be covered in Chapter 2.
- **Configuration Management of Service.** How do you manage the configuration of your microservice so that as it moves between different environments in the cloud you never have to change the core application code or configuration? Managing service configuration will be covered in Chapter 3.
- **Event processing between services**. How do you decouple your microservice using events so that you minimize hardcoded dependencies between your services and increase the resiliency of your application? The topic of event processing between services will be covered in chapter 7.

1.8.2 Microservice Routing Patterns

The microservice routing patterns cover microservice discovery and routing. In a cloud-based application you might have hundreds of microservices instances running. This means you will need to abstract away the physical IP address of these services and have a single point of entry for service calls so that you can consistently enforce security and content policies for all service calls.

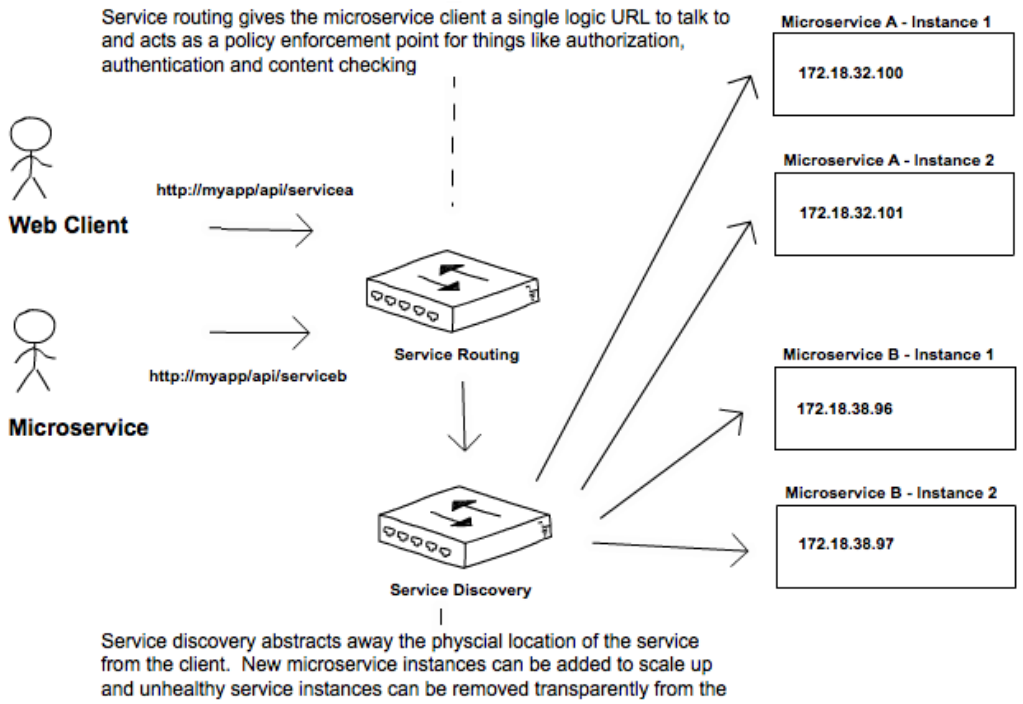

Figure 1.6 – Service discovery and routing are key parts of any large-scale microservice application.

Service discovery and routing really answer the question of how do I get my clients request for a service to a specific instance of a service.

- **Service Discovery**. How do you make your microservice discoverable so client applications can find them without having the location of the service hardcoded into the application? How do you ensure that misbehaving microservice instances are removed from pool of available service instances? Service discovery is covered in Chapter 4.
- **Service Routing** – How do provide a single entry point for all of your services so that security policies and routing rules are applied uniformly to multiple services and service instances in your microservice applications? How do you ensure that each developer in you team do not have to come up with their own solutions for providing routing to their services. Service routing is covered in chapter 6.

While in Figure 1.3 appear to have a hard-coded sequence between them (first comes service routing and the service discovery) to implement one pattern does not require the other. For instance, we can implement service discovery without service routing. Service routing can be implemented without service discovery (even though its implementation is more difficult)

1.8.3 Microservice Client Resiliency Patterns

Since microservices architectures are highly distributed we have to be extremely sensitive in how we prevent a problem in single service (or service instance) from cascading up and out to the consumers of the service. To this end, we are going to cover four topics with client resiliency patterns.

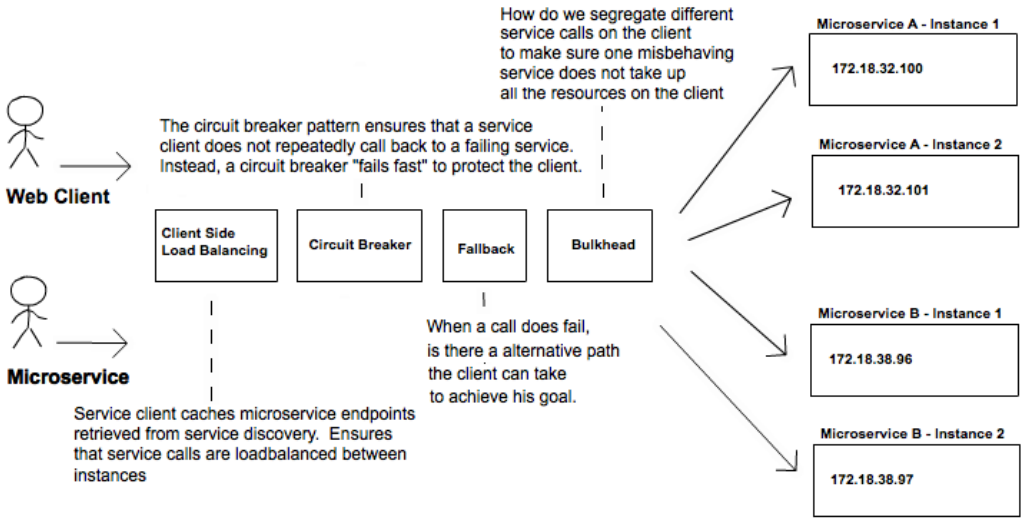

Figure 1.7 – With microservices care must be taken to protect the call from a poorly behaving services

- **Client-side load balancing.** How do we cache the location of our service instances on the service client so that calls to multiple instances of a microservice are load balanced to all of the health instances of that microservice?
- **Circuit Breakers Pattern.** How do you prevent a client from continuing to call a service that is failing or suffering performance problems? When a service is running slow it consuming resources on the client calling it. We want failing microservice calls to fail fast.
- **Fallback Pattern.** When a service call fails how do we provide a "plug-in" mechanism that will allow the service client to try and carry out its work through some alternatives means other then the microservice being called.
- **Bulkhead Pattern.**Microservice applications use multiple distributed resources to carry out their work. How do we compartmentalize these calls so that the misbehavior of one service call does not negatively impact the rest of the application?

These four topics are covered in Chapter 5.

1.8.4 Microservice Build/Deployment Patterns

One of the core parts of a microservice architecture is that each instance of a microservice should be identical to all of its other instances. We can not allow "configuration drift" (something changes on a server after it has been deployed) to occur as this can introduceinstability in your applications.

A phrase too often said

"I made only one small change on the stage server, but I forgot to make the change in production." The resolution of many down systems when I have worked on critical situations teams over the years has often started with those words from a developer or system administrator. Engineers (and most people in general) operate under the notation of good intent. They don't go to work to make mistakes or bring down systems. Instead they are doing the best they can, but they get busy or distracted. So they tweak something on a server fully intending to go back and do it in all of the environments.

At some later point, an outage occurred and everyone is left scratching their head going what is different between the lower environments in production. I have found that the small size and limited scope of microservice make it the perfect opportunity to introduce the concept of "immutable infrastructure" into an organization: Once a service is deployed, the infrastructure it is running on is never touched again by human hands

In my mind having an immutable infrastructure is critical piece of successfully using a microservice architecture, because you have to be able to guarantee in production that every microservice instance you start for a particular microservice is identical to its brethren.

To this end our goal is to integrate the configuration of our infrastructure right into our build deployment process so that we no longer deploy software artifacts like a Java WAR our EAR to an already running piece of infrastructure. Instead we want to "bake" our microservice and the virtual server image it is running on as part of the build process. Then when our microservice gets deployed, the entire machine image with the server running on it gets deployed.

Figure 1.8 illustrates this process.

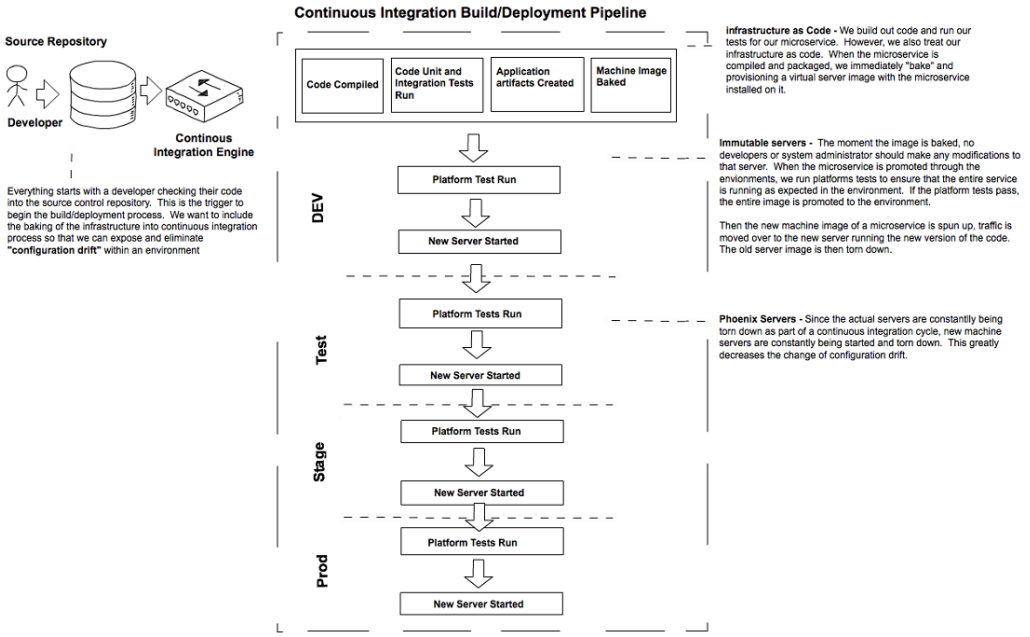

Figure 1.8 We want to deployment of the microservice and the server its running on to be one atomic artifact that is deployed as a whole between environments.

At the end of the book we are going to look at how we change our build and deployment pipeline so that our microservices and the servers they run on are deployed as a single unit of work. In chapter 8 we are going to cover the following patterns and topics:

- **Build and Deployment Pipeline**. How do you create a repeatable build and deployment process that emphasis one button builds and deployment to any environment in your organization?
- **Infrastructure as Code.** How do you treat the provisioning of your services as code that can be executed and managed under source control?
- **Immutable Servers.** Once a microservice image is created how do you ensure that it is never changed after it has been deployed?
- **Phoenix Servers.** The longer a server is running the more opportunity for configuration drift. How do we ensure that our servers that run our microservices get torn down on a regular basis and recreated off an immutable image?

Our goal with these patterns and topics is to ruthlessly expose as quickly as possible and stamp out configuration drift before it is allowed to hit our environment upper environments like stage or production.

1.9 Leveraging Spring Cloud in building your microservices

In this section, I briefly introduce the Spring Cloud technologies that we are going to use as we build out our microservices. This is just a high-level overview; when we use each technology in this book, I'll teach you the details on each as needed.

Implementing all of the patterns above from scratch would be a tremendous amount of work. Fortunately, for us the Spring team has integrated a wide number of battle-tested open-source projects into a Spring sub-project collectively known as Spring Cloud. (http://projects.spring.io/spring-cloud/).

Spring Cloud wrappers the work of open-source companies like Pivotal, HashiCorp and NetFlix in delivering the above patterns. Spring Cloud simplifies setting up and configuring of these projects into your Spring application so that you can focus on writing code and not getting buried in the details of configuring all of the infrastructure that can go with building and deploying a microservice application.

Figure 1.8 below maps the patterns listed in the previous section to the Spring Cloud projects that implement them.

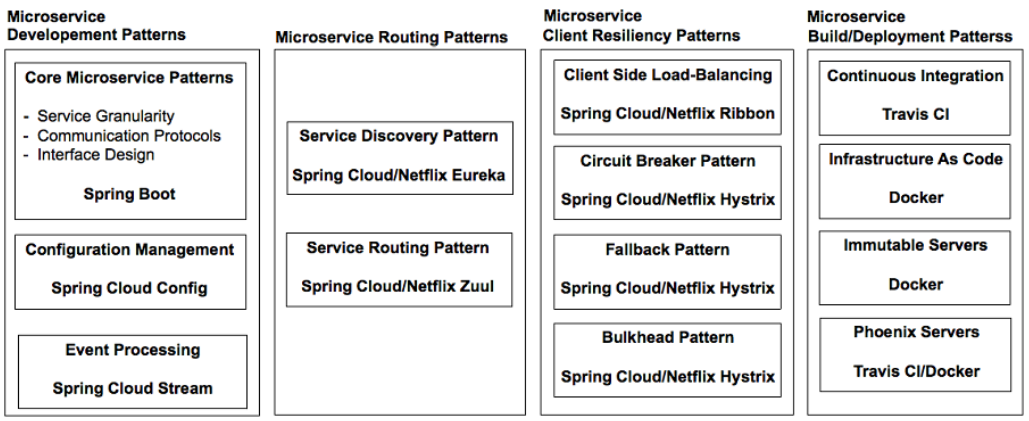

Figure 1.9 Mapping the technologies we are going to use to implement the microservice patterns

Let's walkthrough these technologies in greater detail.

1.9.1 Spring Boot

Spring Boot is the core technology used in our microservice implementation. Spring Boot greatly simplifies microservice development by simplifying the core tasks of building REST-based microservices. Spring Boot greatly simplifies mapping HTTP-style verbs actions to URLs and the serialization of the JSON (JavaScript Object Notation) protocol to and from Java objects. It also simplifies the mapping of Java exceptions back to standard HTTP error codes.

1.9.2 Spring Cloud Config

Spring Cloud Config handles the management of application configuration data through a centralized service so your application configuration data (particularly your environment specific configuration data) is cleanly separated from your deployed microservice. This ensures that no matter how many microservice instances you bring up they will always have the same configuration. Spring Cloud Config has it owns property management repository, but also integrates with open source projects like Git (https://git-scm.com/) , Consul (https://www.consul.io/) and Eureka project (https://github.com/Netflix/eureka).

1.9.3 Spring Cloud Service Discovery

Spring Cloud service discovery allows you to abstract away the physical location (IP and/or server name) of where your servers are deployed from the clients consuming the service. Service consumers invokes business logic for the servers through a logical name rather then a physical location. Spring Cloud Service Discovery also handles the registration and deregistration of services instances as they are started up and shutdown. Spring Cloud Service Discovery can be implemented using Consul (https://www.consul.io/) and Eureka project (https://github.com/Netflix/eureka)as it service discovery engine.

1.9.4 Spring Cloud/Netflix Hystrix and Ribbon

Spring Cloud heavily integrates with Netflix open source projects. For microservice client resiliency patterns, Spring Cloud wrappers the Netflix Hystrix (https://github.com/Netflix/Hystrix)and Ribbon (https://github.com/Netflix/Ribbon)projects and makes leveraging them from within your own microservices trivial to implement. The Netflix Hystrix libraries allow you quickly implement service client resiliency patterns like the circuit breaker and bulkhead patterns.

While the Netflix Ribbon project simplifies integrating with service discovery agents like Eureka, it also provides client-side load-balancing of service calls from a service consumer. This allows a client to continue making service calls even if the service discovery agent is temporarily unavailable.

1.9.5 Spring Cloud/Netflix Zuul

Spring Cloud leverages the Netflix Zuul project (https://github.com/Netflix/zuul)to provide service routing capabilities for your microservice application. Zuul proxies service request and makes sure that all calls to your microservices go through a single "front door" before the actual requests hits your service instance. This centralization of service calls with Zuul allows you to enforce standard service policies like a security authorization authentication, content filtering and routing rules.

1.9.6 Spring Cloud Stream

Spring Cloud Stream (https://cloud.spring.io/spring-cloud-stream/) is an enabling technology that allows you easily integrate lightweight message processing into your microservice. By using Spring Cloud Stream you can build intelligent microservices that can leverage asynchronous events as they occur in your application. Spring Cloud Stream allows to quickly integrate your microservices with AMQP-compliant (https://www.amqp.org/)message brokers like RabbitMQ (https://www.rabbitmq.com/)and Kafka.(http://kafka.apache.org/)

1.9.7 What about Provisioning?

For the provisioning implementations we are going to make a technology shift. The Spring framework(s) are geared towards application development. The Spring frameworks (including Spring Cloud) do not have tools for creating a "build and deployment" pipeline. To this end we are going to leverage the following tools: Travis CI (https://travis-ci.org) for our build tool and Docker (https://www.docker.com/) to build the final server image containing our microservice.

1.10 Spring Cloud by example

In the last section we walked through all the different Spring Cloud technologies that we are going to use as we build out our microservices. Since each of these technologies are independent services it is obviously going to take more than one chapter to explain all of them in detail. However, as I wrap up this chapter, I want to leave you with a small code example that again demonstrates how easy it is to integrate these technologies in your own microservice development effort.

Unlike the first code example in listing 1.1, you will not be able to simply run this code example, as there are a number of supporting services that need to be setup and configured to be leveraged. Don't worry though, the setup costs for these Spring Cloud services (configuration service, service discovery) are a one-time cost in terms of setting up the service. Once they are setup, your individual microservices can leverage these capabilities over and over again. We just get fit all that goodness into a single code example at the beginning of the book.

In the code showing in listing 2.2, I am going to quickly demonstrate how we integrated the service discovery, the circuit breaker, bulkhead and client-side load balancing of remote services into our "Hello World" example.

Listing 1.2 Our Hello World Service using with Spring Cloud

```
package com.thoughtmechanix.simpleservice;

//Removed other imports for conciseness
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
```

```
import   org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

@SpringBootApplication
@RestController
@RequestMapping(value="hello")
@EnableCircuitBreaker                                          (1)
@EnableEurekaClient                                            (2)
public class Application {

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

    @HystrixCommand(threadPoolKey = "helloThreadPool")         (3)
public String helloRemoteServiceCall(String firstName,
                                              String lastName){
        ResponseEntity<String> restExchange =
               restTemplate.exchange(
                    "http://nameservice/name/{firstName}/{lastName}", (4)
                       HttpMethod.GET,
                       null, String.class, firstName, lastName);

       return restExchange.getBody();


    }

@RequestMapping(value="/{firstName}/{lastName}",
       method = RequestMethod.GET)
    public String hello( @PathVariable("firstName") String firstName,
                         @PathVariable("lastName") String lastName) {
       return helloRemoteServiceCall(firstName, lastName)
}
}
```

Listing 1.2 Our simple service leveraging Spring Cloud Capabilities

1. **Enables the service to use the Hystrix and Ribbon Libraries**
2. **Tells the service that it should register itself with a Eureka service discovery agent and that service calls are to use Service discovery to "lookup" the location of remote services**
3. **Wrappers calls to the helloRemoteServiceCall method with a Hystrix Circuit Breaker.**
4. **Uses a decorated `RestTemplate` class to take a "logical" service id and use Eureka under the covers to look up the physical location of the service.**

There is a lot packed into the code example above. Let's quick walkthrough it. Keep in mind that listing 1.2 is for examples purposes only. We will structure and break apart our Spring Boot microservices into a more structured fashion as we progress through the book.

The first thing you should notice as you look at listing 1.2, is the `@EnableCircuitBreaker` and `@EnableEurekaClient` annotations. The `@EnableCircuitBreaker` annotation tells our Spring microservice that we are going to use the Netflix Hystrix libraries in our application. The `@EnableEurekaClient` annotation tells our microservice to register itself with a Eureka Service Discovery agent **and** that we are going to use service discovery to look up remote

REST services endpoints in our code. Note, there is some configuration going on in a property file that will tell our simple service the location and port number of a Eureka server to contact.We first see Hystrix being used when are declaring our hello method.

```
@HystrixCommand(threadPoolKey = "helloThreadPool")
public String helloRemoteServiceCall(String firstName, String lastName)
```

The `@HystrixCommand` annotation is doing two things. First, any time the `helloRemoteServiceCall` method, the method will be invoked on a thread pool managed by Hystrix. If the calls takes too long (default is 1 second), Hystrix will step in and interrupt the call. This is the implementation of the circuit breaker pattern. The second thing this annotation does is create a thread pool called "helloThreadPool" that is managed by Hystrix. All calls to `helloRemoteServiceCall` method will only occur on this thread pool and will be isolated from any other remote service calls being made.

The last thing to note is what is occurring inside the the `helloRemoteServiceCall` method. The presence of the `@EnableEurekaClient` has told Spring Boot that we are going to use to a modified `RestTemplate` class (this is not how the Standard Spring `RestTemplate` would work out of the box) whenever we make a REST service call. This `RestTemplate`class will allow us to pass in a logical service id for the service we are trying to invoke.

```
ResponseEntity<String> restExchange =
        restTemplate.exchange(http://logical-serviceid/name/{firstName}/{lastName}
```

Under the covers, the `RestTemplate` class will contact the Eureka service and lookup the physical location of one or more the "name" service instances. As a consumer of the service, our code never has to nowhere that service is located.

Also `RestTemplate` class above is leveraging Netflix's Ribbon library. Ribbon will retrieve a list of all of the physical endpoints associated with a service and the everytime the service is called by the client, will round robin to the different service instances on the client without having to go through a centralized load balancer. By eliminating a centralized load-balancer and moving it to the client you eliminate another failure point (load balancer going down) in your application infrastructure.

I hope at this point you are impressed because we have added a significant amount of capabilities to our microservice with only a few annotations. That is the real beauty behind Spring Cloud. You as a developer get to take advantage of battle-hardened microservice capabilities from premier cloud companies like Netflix and Consul. These capabilities, if be being used outside of Spring Cloud can be complex and obtuse to setup. Spring Cloud simplifies their use down to literally nothing more then a few simple Spring Cloud annotations and configuration entries.

1.11 Making sure our examples are relevant

I want to make sure this book provides examples that you can relate to as you go about your day-to-day job. To this end, I am going to structure the chapters in this book and the

corresponding code examples around the adventures (misadventures) of a fictitious company called ThoughtMechanix. The examples in this book will not build the entire ThoughtMechanix application. Instead we build some specific microservices from the problem domain at hand and then build the infrastructure that will support these services

ThoughtMechanix is a software development company, whose core product, EagleEye provides an enterprise grade Software Asset Management application. It provides coverage for all of the critical elements: Inventory, Software Delivery, License management, Compliance, Cost and Resource management. Its primary goal is to enable organizations to gain an accurate point in time picture of its software assets.

The company is approximately 10 years old and while they have been experiencing solid revenue growth internally they have been debating about whether or not they should be re-platforming their core product from a monolithic,"on premise" based application or move their application to the cloud. The re-platforming involved with EagleEye can be a "make or break" moment for a company. The company is looking at rebuilding their core product EagleEye on a new architecture. While much of the business logic for the application will remain in place, the application itself will be broken down from a monolithic design to a much smaller microservice design whose pieces can be deployed independently to the cloud.

The ability to successfully adopt cloud-based, microservice architecture is going to impact all parts of a technical organization. This includes the architecture, engineering, testing and operations teams. Input is going to be needed from each group and in the end they are probably going to see the need for some reorganization as the team re-evaluates their responsibility in this new environment. Let's begin our journey with Thoughtmechanix as we begin the fundamental work of identifying and building out some of the microservices used in "Eagle Eye" and then building these services using Spring Boot.

1.12 Summary

- Microservices are extremely small pieces of functionality that are responsible for one specific area of scope.
- There are no industry standards around microservices. Unlike other early web service protocols, microservices take a principle based approach and align with the concepts of REST and JSON.
- Writing microservices are easy, but fully operationalizing them for production requires additional forethought. We introduced several categories of microservice development patterns including: core development, routing patterns, client resiliency and build/deployment patterns.
- While microservices are language agnostic, we introduced two Spring frameworks that significantly help in building microservices: Spring Boot and Spring Cloud.
- Spring Boot is used to simplify the building of REST-based/JSON microservices. Its goal is to allow you to build microservices quickly with nothing more then a few annotations.

- Spring Cloud is a collection of open-source technologies from companies like NetFlix and HashiCorp that have been "wrappered" with Spring annotations to significantly simplify the setup and configuration of these services.

2

Building microservices with Spring Boot

This chapter covers

- The key characteristics of a microservice
- How microservices fit into an overall cloud architecture
- How to decompose a business domain into a set of microservices
- Implementing a simple microservice using Spring Boot
- Understanding the architectural, development and operational perspectives needed to build microservice-based applications
- When not to use microservices

The history of software development is littered with the tales of large development projects that after millions of dollars in cost, hundreds of thousands of software developer hours and with some of the best and brightest minds in the industry working on them somehow never managed to deliver anything of value to their customers and literally collapsed under the complexity and overall weight of the project.

These mammoth projects tended to follow large traditional waterfall methodologies that insisted that all of the requirements and design for the application being built be defined at the beginning of the project. So much emphasis was placed on getting all of the specifications for the software "correct" that there was very little leeway to meet new business requirements or refactor and learn from mistakes made in the early stages of development.

The reality though is that software development is not a linear process of definition and execution, but rather an evolutionary one where it takes several iterations of **communicating**

with, learning from and delivering to the customer before the development team really understands the problem at hand.

Compounding the challenges of using traditional, waterfall methodologies is that many times the granularity of the software artifacts being delivered in these type of projects are:

- **Tightly coupled**. The invocation of business logic happens at the programming language level instead of through implementation neutral protocols like SOAP or REST. This greatly increases the chance that even a small change to an application component can break other pieces of the application and introduce new bugs.
- **Leaky**. Most large software applications will manage different types of data in their application. For instance, a Customer Relationship Management (CRM) application might manage customer, sales and product information. In a traditional model, this data is kept in the same data model and within the same datastore. Even though there are obvious boundaries between the data, too often it is tempting for a team from one domain to directly access the data that belongs to another team.

  This easy access to this data creates hidden dependencies and allows implementation details of one component's internal data structures to leak throughout the entire application. This means that even small changes to a single database table can require a significant amount of code changes and regression testing throughout the entire application.
- **Monolithic.** Since most of the application components for a traditional application reside in a single code-base that is shared across multiple teams, any time a change to the code is made, the entire application has to be recompiled, re-run through an entire testing cycle and redeployed. Even small changes to the application's code base, whether they are new customer requirements or bug fixes become expensive and time consuming while large changes become nearly impossible to do in a timely fashion.

A microservice-based architecture takes a different approach to delivering functionality. Specifically, microservice-based architectures are

- **Constrained.** Microservices have only a single set of responsibilities and are very narrow in scope. Microservices embraces the Unix philosophy that an application is nothing more than a collection of distributed services that should do "One thing and one thing well."
- **Loosely coupled.** Microservice-based applications are a collection of small services that only interact with one another through a non-implementation specific interface using a non-proprietary invocation protocol (e.g. HTTP and REST). As long as the interface for the service does not change, the owners of the microservice have more freedom to make modifications to the service then in a traditional application architecture.
- **Abstracted.** Microservices completely own their data structures and data sources. Data owned by a microservice can only be modified by that service. Access control to the

database holding the microservice's data can be locked down to only allow the service access to it.

- **Independent.** Each microservice in a microservice application can be compiled and deployed independently of the other services used in the application. This means changes can be isolated and tested much more easily than with then a more heavily interdependent, monolithic application.

Why are the microservice architecture attributes listed above important to cloud-based development? Cloud-based applications in general

- **Have a very large and very diverse user-base.** Different customers want different features and they do not want to have wait for a long application release cycle before they can start using these features. Microservices allow features to be delivered quickly because service is small and scope and accessed through a well-defined interface.
- **Have extremely high up-time requirements.** Because of the decentralized nature of microservices, microservice-based applications can more easily isolate faults and problems to specific parts of an application without taking down the entire application. This reduces overall downtime for applications and makes them more resilient to problems.
- **Have uneven volume requirements.** Traditional applications deployed within the four walls of corporate data center will usually have very consistent usage patterns that emerge over time. This makes capacity planning for these types of applications simple. However, in a cloud-based application, a simple tweet on Twitter or a post on Slashdot can drive demand for a cloud-based application through the roof.

  Since microservices application are broken down into small components that can be deployed independently of one another, it is much easier to focus on the components that are under load and scale those components horizontally across multiple servers in a cloud.

This chapter provides you the foundation you need to target and identify microservices in your business problem, build the skeleton of a microservice and then understand the operational attributes that need to be in place for a microservice to be deployed and managed successfully in production.

To successfully design and build microservices you need to approach microservices as if you were a police detective interviewing witnesses to a crime. Even though every witness sees the same events take place, their interpretation of the crime is going to shaped by their background, what was important to them (for example, what motivates them) and what environment pressures were brought to bear at that moment they witnessed the event. Each participant has their own perspective (and biases) of what they consider important.

Like a successful police detective trying to get to the truth, the journey to build a successful microservice architecture involves incorporating the perspective of multiple individuals within your software development organization. While it takes more than just

technical people to a deliver an entire application, the author believes that the foundation for successful microservice development starts with the perspectives of three critical roles:

- **The Architect.** Whose job it is to see "the big picture" and understand how an application can be decomposed down into individual microservices and how the microservices will interact to deliver a solution.
- **The Software Developer.** The person who actually writes the code and understands in detail how the language and development frameworks for the language will be used to deliver a microservice.
- **The DevOps Engineer.** The individual who brings the intelligence to how the services are deployed and managed throughout not only production, but also throughout all of the non-production environments. The watchword for the DevOps Engineer is consistency and repeatability in every environment.

In this chapter we are going to demonstrate how to design and build a set of microservices from the perspectives of each of these roles. We will do this using Spring Boot and Java with the goal being that by the time we are done with the chapter we will have a service that can be packaged and deployed to the cloud.

2.1 The architect's story: designing the microservice architecture

An architect's role on a software project is to provide a working model of the problem that is trying to be solved. The job of the architect is to provide the scaffolding that the developers will build their code against so that all of the pieces of the application "just" fit together.

When building a microservices architecture, a project's architect is going to be focused on three key tasks:

1. Decomposing the business problem
2. Establishing service granularity
3. Defining the service interfaces

2.1.1 Decomposing the business problem

In the face of complexity, most people will try to break the problem they are working on into manageable chunks. They do this so they don't have to try and fit all the details of the problem in their head. Instead, they abstract the problem down to a few key parts and then look for the relationships that exist between these parts.

In a microservices architecture, the architect is going to break the business problem into "chunks" that represent discrete domains of activity. These chunks will encapsulate the business rules and the data logic associated with a particular part of the business domain.

While you want microservices to encapsulate all of the business rules for carrying out a single transaction, this is not always feasible. You will often have situations where you need to have groups of microservices working across different parts of the business domain to

complete an entire transaction. An architect will tease apart the service boundaries of a set of microservices by looking at where the data domain does not seem to fit together.

For example, an architect might look at a business flow that is to be carried out by some code and realize that he or she needs both customer and product information. The presence of two discrete data domains is a good indication that there are multiple microservices at play. How the two different parts of the business transaction interact usually become the service interface for the microservices.

Breaking apart a business domain is an art form rather than a black and white science.

The author likes to use the following guidelines for identifying and decomposing a business problem into microservice candidates:

1. **Describe the business problem and listen to the nouns you are using to describe the problem.** If you find yourself using the same nouns over and over in describing the problem, that is usually an indication of a core business domain and an opportunity for a microservice.
2. **Pay attention to the verbs.** Verbs highlight actions and often represent the natural contours of a problem domain. If you find yourself saying thing like "transaction X needs to get data from thing A and thing B" that usually indicates that there are multiple services at play here.
3. **Look for data cohesion**. As you break apart your business problem into discrete piece, look for data that is highly related to one another. If all the sudden during the course of your conversation you are reading or updating data that is radically different from what you have been discussed so far, you potentially have another service candidate. **Microservices should completely own their data.**

So let's take the guidelines above and apply them to a real world problem. In chapter 1, we introduced an existing software product called Eagle Eye that is used for managing software assets like software licenses and SSL (secure socket layer) certificates. These items are deployed to various servers throughout an organization.

Eagle Eye is a traditional monolithic web application that is deployed to a J2EE application server residing within a customer's data center. Our goal is to tease apart the existing monolithic application into a set of services.

We are going to start by interviewing all of the users of the Eagle Eye application and talk to them about how they interact and use Eagle Eye. The diagram below captures a summary of the conversations we might have with the different business customers.

By looking at how the users of Eagle Eye interact with the application and how the data model for the application is broken out we could break out the following problem domains and microservice candidates:

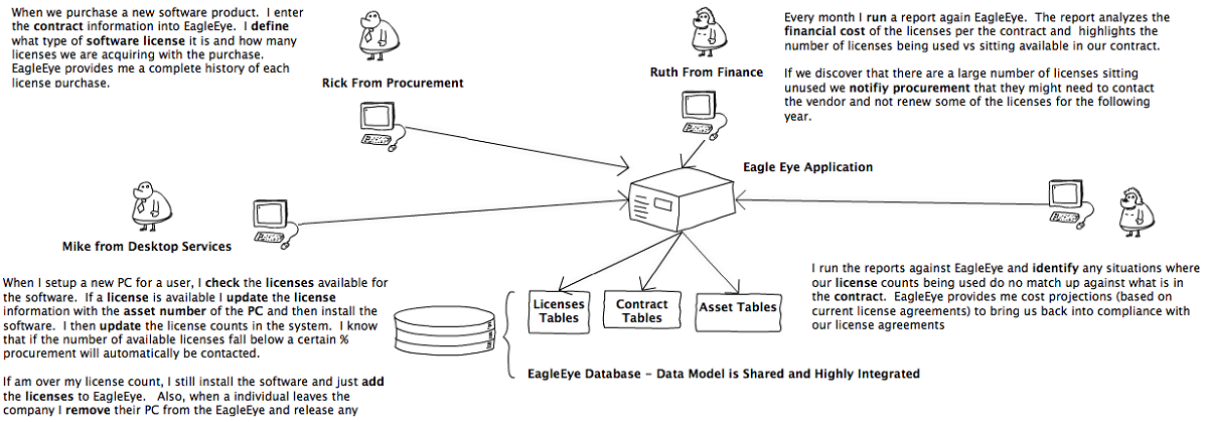

Figure 2.1 Interview the Eagle Eye users and understand how they do their day-to-day work

As you can see above we have highlighted a number of nouns and verbs that have come up during our conversation with the business users. Since this is an existing application, we are going to take a look at the applications data model and map the major nouns back to tables in the physical data model. In an existing application you might have hundreds of tables, but each table will usually map back to single set of logical entities. In the diagram below, we offer a simplified data model based on the conversations we had with the eagle eye customers and the data model that is already in placeof the different microservice candidates. Based on the business interviews and the data model above our microservice candidates are: organization, license, contract and assets service.

Once we have a simplified data model this will usually give you a good clue

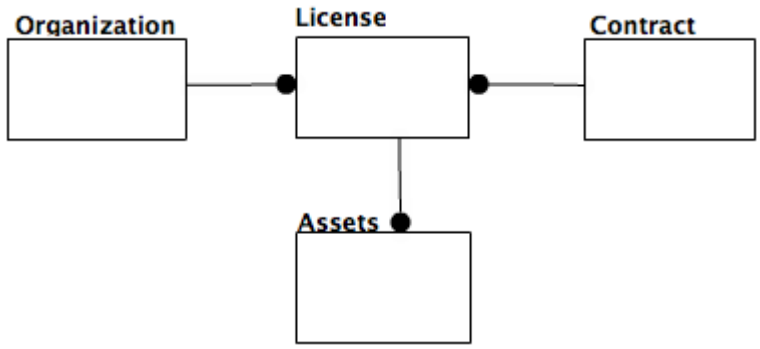

Figure 2.2 – A simplified eagle eye data model

2.1.2 Establishing Service Granularity

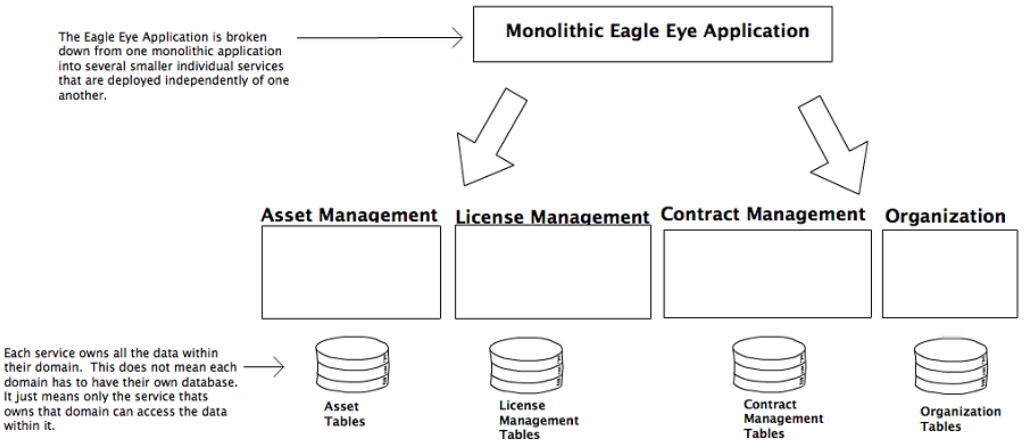

Figure 2.3 Our target microservices for the eagle eye problem domain

After a problem domain has been broken down into discrete pieces, you will often find yourself struggling with whether or not you have achieved the right level of granularity for your services. A microservice that is too coarse-grained or fine-grained will have a number of telltale attributes that will be discussed shortly.

When building a microservice architecture, the question of granularity is important, but there a few concepts you can use to guide yourself to the right solution:

1. **It's better to start broad with your microservice and refactor to smaller services.** It's easy to go overboard when you first begin your microservice journey and have everything be a microservice. However, decomposing your problem domain into very small services can often lead to premature complexity as your microservices devolve into nothing more than very fine-grained data services.
2. **Focus first on how your services will interact with one another.** This will help establish the coarse-grained interfaces of your problem domain. It is easier to refactor from being too coarse-grained to being too small grained.
3. **Service responsibilities will change over time as your understanding of the problem domain grows.** Often times a microservice will grow in responsibilities as new functionality is requested out for the application. What starts as a single microservice might grow into multiple services with the original microservice acting as orchestration layer for these new services and encapsulating the functionality of these new services from other parts of the application.

The smells of a bad microservice

How do you know if you do not have the right size for your microservices?If a microservice is too coarse-grained you are likely to see:

1) **A service with too many responsibilities**. The general flow of the business logic in the service is complicated and seems to be enforcing an overly diverse number of business rules.
2) **The service is managing data across a large number of tables**. A microservice is the system of record for the data it manages. If you find yourself persisting data to multiple tables or reaching out to tables outside of the immediate database, this is a clue the service is too big. The author likes to use a guideline of no more than 3-5 tables should be owned by microservice. Any more and your service is likely to have too much responsibility.
3) **Too many test cases**. Services can grow in size and responsibility over time. If you have a service that started with a small number of test cases and ends up with hundreds of unit and integration test cases, you might be looking at a service that needs to be refactored.

What about a microservice that are too fine-grained?

1) **The number of microservices in one part of the problem domain breed like rabbits.** If everything becomes a microservice, composing business logic out of the services becomes complex and difficult as the number of services needed to get a piece of work done grows tremendously.
   A common smell is when you have dozens of microservices in an application where each service interacts with only a single database table.
2) **Your microservices are heavily interdependent on one another.** You find that the microservices in one part of your part of the problem domain keep calling back and forth between each other to complete a single user request.
3) **Become a collection of simple CRUD services.** Microservices are expression of business logic and not just an abstraction-layer over your data sources. If your microservices do nothing but CRUD related logic they are probably too fine-grain.

Microservices architecture should be developed with an evolutionary thought process where you know that you are not going to get the design right the first time. That is why it is better to start with your first set of services being more coarse-grained then fine-grained. It is also important not to be dogmatic with your design. There will be times when you will run into physical constraints on your services where you will need to make an "aggregation" service that joins data together because two separate services will be too "chatty" or where there are no clear boundaries between the domain lines of a service.

In the end, take a pragmatic approach and deliver rather than spending too much time trying to get the design perfect and have nothing to show for your effort.

2.1.3 Talking to one another: service interfaces

The last part of the of the architect's input is about defining how the microservices in your application are going to talk with one another. When building business logic with microservices, the interfaces for the services should be intuitive and developers should get a rhythm of how all the services work in the application by just learning one or two of the services in the application.

In general, the following guidelines can be used for thinking about service interface design.

1. **Embrace the REST philosophy**. The REST approach to services is at heart the embracing of HTTP as the invocation protocol for the services and the use of e standard HTTP verbs (GET, PUT, POST and DELETE). Model your basic behaviors around these HTTP verbs.
2. **Use URI's to communicate intent.** The URI you use as endpoints should for the service should describe the different resources in your problem domain and also provide a basic mechanism for relationships of resources within your problem domain.
3. **Leverage JSON for your requests and responses.** JavaScript Object Notation (in other words, JSON) is an extremely lightweight data serialization protocol and much easier to consume then XML.
4. **Use HTTP Status Codes to communicate results.** The HTTP protocol has a rich body of standard response codes to indicate the success or failure of a service. Learn these status codes and most importantly use them consistently across all of your services.

All of the basic guidelines drive to one thing, making your service interfaces easy to understand and consumable. You want to be able to have a developer sit down and look at the service interfaces and just start using them. If a microservice is not easy to consume, developers will go out of their way to work around and subvert the intention of the architecture.

2.2 When not to use microservices

We have spent this chapter talking about why microservices are a powerful architectural pattern for building applications. However, we have not touched upon when we should **not** use microservices to build your applications. Let's walkthrough them:

1. Complexity building distributed systems
2. Virtual Server/Container Sprawl
3. Application Type
4. Data Transactions and Consistency

2.2.1 Complexity of building distributed systems

Because microservices are distributed and fine-grained (small) the introduce a level of complexity into your application that would not be there in more monolithic applications. Microservice architectures require a high degree of operational maturity. Do not consider using microservices unless your organization is willing to invest in the automation and operational (monitoring, scaling) that a highly distributed application needs to be successful.

2.2.2 Server sprawl

One of the most common deployment models for microservices is to have one microservice instance deployed on one server. This means that in a large microservices-based application, I might end up with 50-100 servers or containers (usually virtual) that have to be built and

maintained just in production alone. Even with the lower cost of running these services in the cloud, the operational complexity of having to manage and monitor these servers can be tremendous.

The flexibility of microservices has to be weighed against the cost of running all of these servers.

2.2.3 Type of application

Microservices are geared towards reusability and are extremely useful for building large applications that need to be highly resilient and scalable. This is one of the reasons why so many cloud-based companies have adopted microservices. If you are building small, departmental level applications or applications with a very small user base, the complexity associated with building on a distributed model like microservices might be more expense then it is worth.

2.2.4 Data Transformations and consistency

As you begin looking at microservices, you need to think through the data usage patterns of your services and how service consumers are going to use them. A microservice "wrappers" a small number of tables and work well as mechanism for performing "operational" tasks like creating, adding and performing simple (non-complex) queries against a store.

If your applications need to do complex data aggregation or transformation across multiple sources of data, the distributed nature of microservices will make this work difficult. Your microservices will invariably take onto much responsibility and also are vulnerable to performance problems.

Also keep in mind that there is no standard for performing transactions across microservices. This means that you if you need transactions, you will need to build that yourself. In addition, as you will see in Chapter 7, microservices can communicate amongst themselves using messages. Messaging introduces latency in data updates. Your applications need to be able to handle eventual consistency where updates that are applied to your data might not immediately appear.

2.3 The developer's tale: building a microservice with Spring Boot and Java

When building a microservice, moving from the conceptual space to the implementation space requires a shift in perspective. Specifically, as a developer you need to establish a basic pattern of how each of your microservices in your application is going to implemented. While each service is going to be unique, you want to make sure that you are using a framework that removes boilerplate code and that each piece of your microservice is laid out in the same consistent fashion.

In the section of the application we are going to explore the developer's perspective by building the licensing microservice from our Eagle Eye domain model. Our licensing service is going to be written using Spring Boot. Spring Boot is an abstraction layer over the standard Spring libraries that allows developer to very quickly build Groovy and Java-based web applications and microservices with significantly less ceremony and configuration as a full-blown Spring application.

For our licensing service example, we are going use Java as our core programming language and Apache Maven as our build tool.

Over the next several sections we are going to:

1. Build the basic skeleton of the microservice and a maven script to build the application.
2. Implement a Spring bootstrap class that will start the Spring container for the microservice and initiate the kick off of any initialization work for the class.
3. Implement a Spring Boot controller class for mapping an Endpoint to expose the endpoints of the service.

2.3.1 Getting started with the skeleton project

To begin, we are going to create a skeleton project for the licensing. You can either pull down the source code down from github[3] or create a licensing-service project directory with the following directory structure:

licensing-service

- src/main/java/com/thoughtmechanix/licenses
  - config
  - controllers
  - model
  - repository
  - services
- resources

Once this directory structure is pulled down or created let's begin by writing out our maven script for the project. This will be the `pom.xml` file located at the root of the project directory. Listing 2.1 shows the maven POM file for our licensing service.

Listing 2.1. Maven pom file for the licensing service

```
<?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
```

[3] https://github.com/carnellj/native_cloud_apps

```
       http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.thoughtmechanix</groupId>
    <artifactId>licensing-service</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>Eagle Eye Licensing Service</name>
    <description>Licensing Service</description>

    <parent>
         <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-parent</artifactId>  (1)
           <version>1.2.4.RELEASE</version>
           <relativePath/>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>  (2)
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>  (3)
        </dependency>
    </dependencies>

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

            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>  (4)
            </plugin>
        </plugins>
    </build>
</project>
```

(1) **1 - Tells Maven to include the Spring Boot Starter Kit dependencies**
(2) **2 - Tells Maven to include the Spring Boot Web dependencies**
(3) **3 - Tells Maven to include the Spring Actuator dependencies**
(4) **4 - Tells Maven to include Spring specific maven plugins for building and deploying Spring Boot applications**

We are not going to go through the entire script in detail, but there are a few key areas to note as we start out. Spring Boot is broken into a large number of individual projects. The philosophy is that you should not have to "pull down the world" if you are going to not use different pieces of Spring Boot in your application. This also allows the various Spring Boot projects to release new versions of code independently of one another. To help simplify the life of the developers, the Spring Boot team has gathered related dependent projects into various

"starter" kits. In step 1 of the maven POM we are telling maven that we need to pull down version 1.2.4 of the Spring Boot framework

In Step's 2 and 3 of the maven file, we are identifying that we are pulling down the Spring Web and Spring Actuator "starter" kits. These two projects are at the heart of almost any Spring Boot REST based service. You will find that as we build more functionality into our services, the list of these dependent projects will become longer.

Also, Spring Source has provided a number of maven plugins that simplify the build and deployment of the Spring Boot applications. Step #4 tells our maven build script to install the latest Spring Boot maven plugin. This plugin contains a number of add-on tasks (such as spring-boot:run) that simplify our interaction between maven and Spring Boot.

Finally, you will see a comment that some sections of the maven file have been removed. For the sake of the trees, I did not include the Spotify Docker plugins in the above list. Every chapter in this book includes Docker files for building and deploying the application as Docker containers. Details of how to build these Docker images can be found in the README.md file found in the code sections of each chapter.

2.3.2 Booting our Spring Boot application: writing the Bootstrap class

Our goal is to get a simple microservice up and running in Spring Boot and then iterate on it to deliver functionality. To this end we need to create two classes in our licensing service microservice:

- A Spring Bootstrap class that will be used by Spring Boot to start up and initialize the application
- A Spring Controller class that will expose the HTTP endpoints that can be invoked on the microservice.

As you will see shortly, Spring Boot leverages annotations to simplify the setup and configuration of the service. This becomes evident as you look at the bootstrap class below. This bootstrap class is located in the `src/main/java/com/thoughtmechanix/licenses/Application.java` file.

Listing 2.2 Introducing the @SpringBootApplication Annotation

```
package com.thoughtmechanix.licenses;

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

@SpringBootApplication                                          (1)
public class Application {
  public static void main(String[] args) {
SpringApplication.run(Application.class, args); (2)
    }
}
```

(1) 1 - @SpringBootApplication tells the Spring Boot framework that this is the bootstrap class for the project
(2) 2 – Call to start the entire Spring Boot service

The first thing to note in this code is the use of the `@SpringBootApplication` annotation. This annotation is used by Spring Boot to tell the Spring container that this class is the source of bean definitions for use in Spring. In a Spring Boot application, you can define Spring Beans by:

1. Annotating a Java class with a `@Component`, `@Service` or `@Repository` annotationtag.
2. Annotate a class with a `@Configuration` tag and then define a constructor method for each Spring bean you want to build with a `@Bean` tag.

The `@SpringBootApplication` annotation under the covers marks the Application class above as a configuration class and then begins auto-scanning all of the classes on the Java class path for other Spring beans.

The second numbered item, number 2, is the `Application` class's `main()` method. In the `main()` method the `SpringApplication.run(Application.class, args);` call starts the Spring container and returns a Spring `ApplicationContext` object. (Note: We are not doing anything with the `ApplicationContext` so it is not shown in the code.)

The easiest thing to remember about the `@SpringBootApplication` annotation and the corresponding Application class is that it is the bootstrap class for the entire microservice. Core initialization logic for the service should be placed in this class.

2.3.3 Building the doorway into the microservice: the Spring Boot controller

Now that we have gotten the build script out of the way and implemented a simple Spring Boot Bootstrap class, we can actually begin writing our first code that will actually do something. This code will be our Controller class. In a Spring boot application, a Controller class is what exposes the services endpoints and maps the data from an incoming HTTP request to a Java method that will process the request.

Give it a REST

All of the microservices in this book will follow the REST approach to building our services. Doing an in-depth discussion of REST is outside of the scope this book[4], but for our purposes all of the services we build will have the following characteristics:

1) **Use HTTP as the invocation protocol for the service**. The service will be exposed via HTTP endpoint and will use the HTTP protocol to carry data to and from the services
2) **Map the behavior of the service to standard HTTP verbs**. REST emphasizes having services map their behavior to the HTTP verbs of POST, GET, PUT and DELETE verbs. These verbs map to the Create, Read, Update and Delete (CRUD) functions found in most services.

[4] Probably the most comprehensive coverage of the design of REST services is Ian Robinson, Jim Webber and Savas Parastatidis book Webber, Jim, Savas Parastatidis, and Ian Robinson. REST in Practice: Beijing: O'Reilly, 2010. Print.

3) **Leverage JSON as the serialization format for all data going to and from the service**. This is not a hard-and fast principle for REST-based microservices, but JSON has become lingua franca for serializing data that is going to be submitted and returned by a microservice. XML can be used, but many REST-based applications are heavy in JavaScript. JSON (e.g. JavaScript Object Notation) is the natural format for serializing and de-serializing data being consumed by Java.
4) **Use HTTP status codes to communicate the status of a service call**. The HTTP protocol has developed a rich set of status codes to indicate the success or failure of a service. REST based services takes advantage of these HTTP status codes and other web based infrastructure like reverse proxies and caches can be integrated with your microservices with relative easy.

HTTP is the language of the web and using HTTP as the philosophical framework for building your service is a key to building services in the cloud.

Our first controller class will be located in `src/main/java/com/thoughtmechanix/licenses/controllers/LicenseServiceController.java`. This class will expose four HTTP endpoints that will map to the POST, GET, PUT and DELETE verbs.

Let's walk through the controller class and look at how Spring Boot provides us with a set of annotations that keeps the effort needed to expose our service endpoints to a minimum and allows us to focus on building the business logic for the service. We will start by just looking at the basic controller class definition without any class methods in it yet. Listing 2.3 shows the controller class that we built for our licensing service.

Listing2.3 Marking the LicenseServiceController as a Spring RestController

```
package com.thoughtmechanix.licenses.controllers;

import …  // Removed for conciseness

@RestController                                    (1)
@RequestMapping(value="v1/organizations/{organizationId}/licenses")    (2)
public class LicenseServiceController {
}
```

(1) 1 - @RestController tells Spring Boot this is a REST-based services and will automatically serialize/deserialize service request/response to JSON
(2) 2 – Exposes all the HTTP endpoints in this class with a prefix of /v1/organizations/{organizationId}/licenses

We are going to begin our exploration by looking at the `@RestController` annotation (1). The `@RestController` is a class-level Java annotation and is used to tell the Spring Container that this Java class is going to be used for a REST-based service. This annotation automatically handles the serialization of data passed into service into the services as JSON or XML (by default the `@RestController` class will serialize returned data into JSON). Unlike the traditional Spring `@Controller` annotation, the `@RestController` annotation does not require you as the developer to return a `ResponseBody` class from your controller class. This is all handled by the presence of the `@RestController` annotation.

The second annotation shown above is the `@RequestMapping` annotation. The `@RequestMapping` annotation can be used as a class level and method level annotation. The `@RequestMapping` annotation is used to tell the Spring container the HTTP endpoint that the service is going to expose to the world. When you use the class level `@RequestMapping` annotation, you are establishing the root of the URL for all of the other endpoints exposed by the controller.

In listing 2.3, the`@RequestMapping(value="v1/organizations/{organizationId}/licenses")` uses the `value` attribute to establish the root of the URL for all endpoints exposed in the controller class. This means that all service endpoints exposed in this controller will start with `/v1/organizations/{organizationId}/licenses` as the root of their endpoint. The `{organizationId}` is a placeholder that indicates we expect the URL to be parameterized with an organizationId passed in every call. The use of organizationId in the URL allows us to differentiate between the different customers who might be using our service.

Now let's add our first method to our controller. This method will implement the GET verb used in a REST call and return a single `License` class instance. (Note: For purposes of this discussions we are going to simply instantiate a Java class called `License`.)

Listing 2.4 Exposing an individual GET HTTP Endpoint

```
@RequestMapping(value="/{licenseId}",method = RequestMethod.GET) ❶
public License getLicenses(
             @PathVariable("organizationId") String organizationId,
@PathVariable("licenseId")      String licenseId) {              ❷
        return new License()
            .withId(licenseId)
            .withProductName("Teleco")
            .withLicenseType("Seat")
            .withOrganizationId("TestOrg");
 }
```

❶ 1 – Creates a GET endpoint with the value v1/organizations/{organizationId}/licenses{licenseId}
❷ 2 – Maps two parameters from the URL (organizationId and licenseId) to method parameters

The first thing we have done in our code from listing 2.4 is annotate the `getLicenses()` method with a method level `@RequestMapping` annotation, passing in two parameters to the annotation: `value` and `method`. With a method-level `@RequestMapping` annotation, we are building on the root-level annotation specified at the top of the class to match all HTTP requests coming to the controller that match `/v1/organizations/{organizationId}/licences/{licensedId}`. The second parameter of the annotation, `method`, specifies the HTTP verb that the method will be matched on. In the example above, we are matching on the GET method as represented by the `RequestMethod.GET` enumeration.

The second thing to note about the code listing 2.4 is that we use the `@PathVariable` annotation in the parameter body of the `getLicenses()` method. (2) The `@PathVariable`

annotationis used to map the parameter values passed in the incoming URL (as denoted by the `{parameterName}` syntax) to the parameters of our method. In our code example from listing 2.4, we are mapping two parameters from the URL, `organizationId` and `licenseId`, to two parameter level variables in the method.

```
@PathVariable("organizationId") String organizationId,
@PathVariable("licenseId")      String licenseId)
```

Endpoint Names Matter

Before you get too far down the path of writing microservices, make sure that you (and potentially other teams in your organization) establish some standards around the endpoints that are going to be exposed via your services. The URLs (Uniform Resource Locator) for the microservice should be used to clearly communicate the intent of the service, the resources the service manages and the relationships that exists between the resources managed within the service. The author has found the following guidelines useful naming service endpoints.

1) **Use clear URL names that establish what resource the service represents.** Having a canonical format for defining URLs will help your API feel more intuitive and easier to use. **Be consistent in your naming conventions.**
2) **Use the URL to establish relationships between resources.** Often times you will have parent-child relationship between resources within your microservices where the child really does not exist outside of context of the parent (hence you might not have a separate microservice for the child). Use the URLs to express these relationships. However, if you find that your URLs tend to be excessively long and nested, then maybe your microservice is trying to do too much.
3) **Establish a versioning scheme for URLS early.** The URL and its corresponding endpoints represent a contract between the service owner and consumer of the service. One common pattern is to pre-pend all endpoints with a version number. Establish your versioning scheme early and stick to it. It's extremely difficult to retrofit versioning to URLS after you already have several consumers using them.

So at this point we have something we can actually call as a service. From a command line window, go to your project directory where you have downloaded the sample code and execute the following maven command:

```
mvn spring-boot:run
```

As soon as you hit the return key, you should see Spring Boot launch an embedded Tomcat server and start listening on port 8080.

Figure 2.4 The Licensing Service starting successfully

Once the service is started we can directly hit the exposed endpoint. Since our first method exposed is a GET call, we can use a number of methods for invoking the service. The author's preferred method is to use browser based tool like Chrome-based plugin POSTMAN or CURL for calling the service.

Figure 2.5 Ourlicensing service being called with POSTMAN

At this point we have a running skeleton of a service. However, from a development perspective this service is not complete. A good microservice design does not eschew segregating the service into well-defined business logic and data access layers. As we progress

in later chapters, we will continue to iterate on this service and deliver in how to further structure this service.

Let's switch to the last perspective where we explore how a DevOps engineer would want to operationalize the service and package it for deployment to the cloud.

2.4 TheDevOps story: building for the rigors of runtime

For the DevOps engineer, the design of the microservice is all about being able to manage the service after it goes into production. Writing the code is often the easy part. Keeping it running is the hard part.

While DevOps is a rich and emerging IT field, we will start our microservice development effort with four principles and build on these principles later on in the book. These principles are

1. A microservice should be a **self-contained**, **independently deployable** with multiple instances of the service being started up and torn down with a single software artifact.
2. A microservice should be **configurable**. When a service instance starts up it should be able to read the data it needs to configure itself from a central location or have its configuration information passed in as environment variables. No human intervention should be required to configure the service.
3. A microservice instance needs to be **transparent** to the client. The client should never know the exact location of a service. Instead, a microservice client should talk to a service discovery agent that will allow the application to locate an instance of a microservice without having to know its physical location.
4. A microservice should be able to **communicate** their health. This is a critical part of our cloud architecture. Microservice instances will fail and clients need to be able to route around bad service instances.

The four principles above expose the paradox that can exist with microservice development. Microservices are smaller in size and scope, but their use introduces more moving parts in an application, especially because microservices are distributed and running completely independently one of another in their own distributed containers. This introduces a high degree of coordination and also more opportunities for failure points in the application.

From a DevOps perspective we must address the operational needs for a microservice upfront and translate the 4 principles listed above into a standard set of lifecycle events that occur every time a microservice is built and deployed to an environment. Our 4 principles can be mapped to following operational lifecycle steps:

- **Service assembly.** How do we package and deploy our service to guarantee repeatability and consistency that the same service code and runtime is deployed exactly the same way?
- **Service bootstrapping.** How do we separate our application and environment-specific configuration code from the run-time code so we can start and deploy a microservice

instance quickly in any environment without human intervention to configure the microservice?

- **Service registration/discovery.** When a new microservice instance is deployed how do we make the new service instance discoverable by other application clients.
- **Service monitoring.** In a microservices environment it is extremely common for multiple instances of the same service to be running due to high-availability needs. From a DevOps perspective we need to monitor microservice instances and ensure that any faults in our microservice are routed around and those ailing service instances are taken down.

Figure 2.6 shows how these four steps fit together.

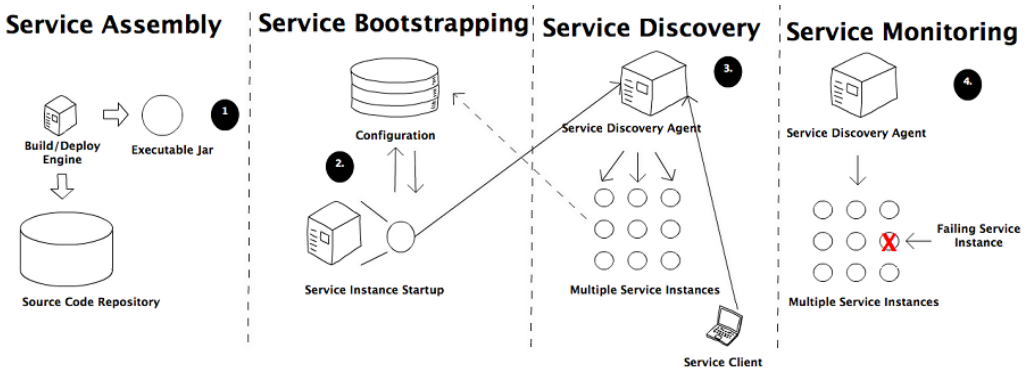

Figure 2.6 The operational microservice lifecycle

2.4.1 Service assembly: packaging and deploying your microservices

From a DevOps perspective, one of the key concepts behind a microservice architecture is that multiple instances of a microservice can be deployed quickly in response to a change application environment (e.g. sudden influx of user requests, problems within the infrastructure, and so on).

In order to accomplish this a microservice needs to be packaged and installable as a single artifact with all of its dependencies defined within it. This artifact can then be deployed to any server with a Java JDK installed on it. These dependencies will also include the run-time engine (e.g., an HTTP server or application container) that will host the microservice.

This process of consistently building, packaging,and deploying are is the service assembly (step 1 in figure 2.6). Figure 2.7 shows additional details about the service assembly step.

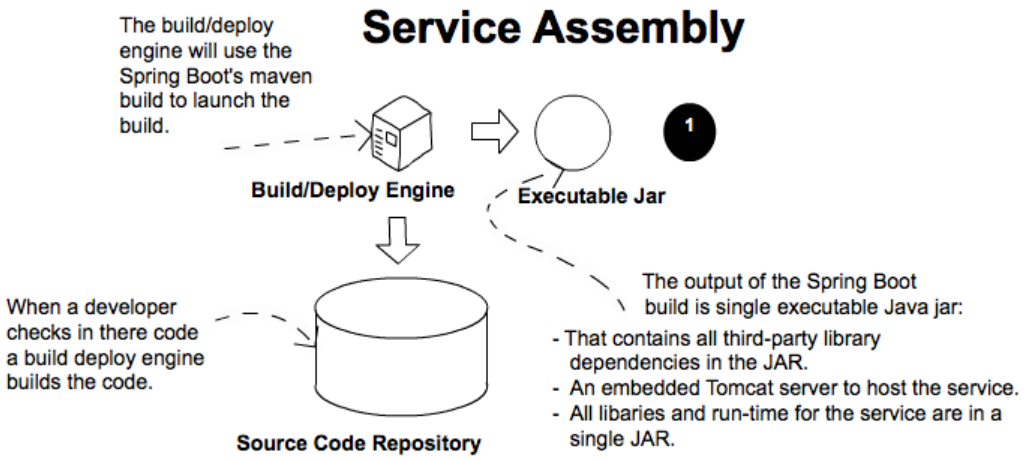

Figure 2.7 In the Service Assembly step, source code is compiled and packaged with its runtime engine

Fortunately, almost all Java microservice frameworks will include a run-time engine that can be packaged and deployed with the code. For instance, in our Spring Boot example above, we can use Maven and Spring Boot to build an executable Java jar file that has an embedded Tomcat engine built right into the jar. In the command-line example below, we are building the licensing service as an executable JAR and then starting the jar file from the command-line.

```
mvn clean package && java –jar target/licensing-service-0.0.1-SNAPSHOT.jar
```

For some operation teams, the concept of embedding a run-time environment right in the JAR file is a major shift in the way they think about deploying applications. In a traditional J2EE enterprise organization, an application is deployed to an application server. This model implies that the application server is an entity in and upon itself and would often be managed by a team of system administrators who would managed the configuration of the servers independently of the applications being deployed to them.

This separation of the application server configuration from the application introduces a number of failure points in the deployment process, because in many organizations the configuration of the application servers is not kept under source control and is managed through a combination of the user interface and home-grown management scripts. It is too easy for configuration drift to creep into the application server environment and suddenly cause what on the surface appeared to be random outages.

The use of a single deployable artifact with the run-time engine embedded in the artifact eliminates many of these opportunities for configuration drift. It also allows you to put the whole artifact under source control and allows the application team to be able to better reason through how their application is built and deployed.

2.4.2 Service Bootstrapping: Managing configuration of your microservices

Service bootstrapping, step 2 in figure 2.6, occurs when the microservice is first starting up and needs to load its application configuration information. Figure 2.8 provides more context around the bootstrapping processing.

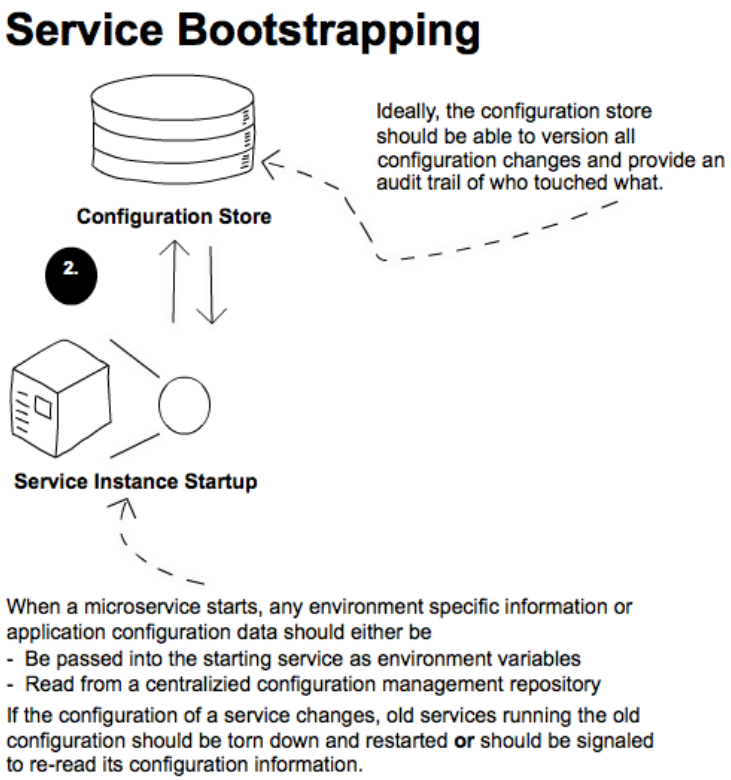

Figure 2.8 As a service starts (boot straps), it reads it configuration from a central repository.

As any application developer knows there will be times when you need to make the run-time behavior of the application configurable. Usually this involves reading your application configuration data from a property file deployed with the application or reading the data out of a datastore like a relational database.

Microservices often run into the same type of configuration requirements. The difference is that in microservice application running in the cloud you might have hundreds or even thousands of microservice instances running. Further complicating this is that the services might be spread across the globe. With a high number of geographically dispersed services it becomes unfeasible to re-deploy your services just to pick up new configuration data.

Storing the data in a data store external to the service solves this problem, but microservices in the cloud offer a set of unique challenge:

1. Configuration data tends to be simple in structure and is usually read frequently and written infrequently. Relational databases are overkill in this situation because they are designed to managed much more complicated data models then a simple set of key-value pairs.
2. Since the data is accessed on a regular basis, but changes infrequently the data has to be readable with a very low level of latency.
3. The data store has to be highly available and close to the services reading the data. A configuration data store can't go down completely, as it would become a single-point of failure for your application.

In Chapter 3 I will show you how manage your microservice application configuration data using things like a simple key-value data store for managing your data.

2.4.3 Service registration and discovery: how clients communicate to your microservices

From a microservice consumer perspective, a microservice should be location transparent because in a cloud based environment, servers are ephemeral. Ephemeral means the servers that a service is hosted on usually have shorter lives then a service running in a corporate data center. Cloud-based services can be started and torn down quickly with an entirely new IP address assigned to the server the services are running on.

By insisting that services are treated as short-lived disposable things, microservice architectures can achieve a high-degree of scalability and availability by having multiple instances of a service running. Service demand and resiliency can be managed quickly as the situation warrants. Each service having a unique and non-permanent IP address assigned to it. The downside to ephemeral services that with services constantly coming up and down, managing a large pool of ephemeral services manually or by hand is an invitation to an outage.

In step 3, service discovery, a microservice instance will need to register itself with the third-party agent. The is process is called service discovery. When a microservice instance registers with a service discovery agent, it will tell the discovery agent two things: the physical IP address or domain address of the service instance and a logical name that an application can use to look up in a service. Some service discovery agents will also require a URL back to the registering service that can be used by the service discovery agent to perform health checks.

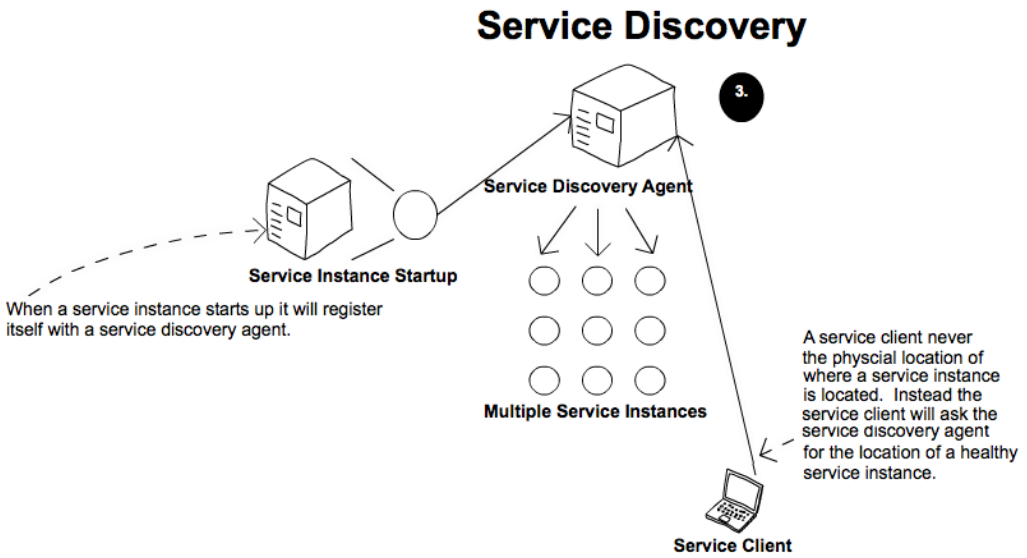

Figure 2.9 A service discovery agent abstracts away the physical location of a service

The service client then communicates with the discovery agent to lookup the actual location of the service.

2.4.4 Communicating a microservice's health

A service discovery agent does not just act as a traffic cop that guides the client to the location of the service. In a cloud-based microservice application, you will often have multiple instances of a service running. Sooner or later one of those service instances will fail. The service discovery agent needs to be able to ascertain the health of all of the service instances registered with it and remove any service instances from its routing tables to ensure that clients are not sent a service instance that has failed.

After a microservice has come up, the service discovery agent will continue to monitor and ping the health check interface to ensure that that service is available. This is step #4 in figure 2.6. Figure 2.10 provides more context around this step.

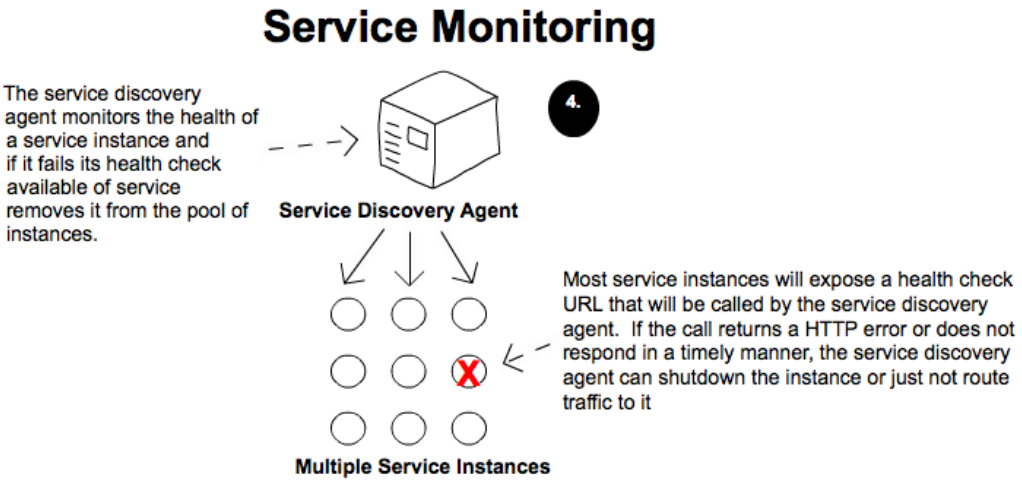

Figure 2.10 The service discovery agent uses the exposes health URL to check microservice health

By building a consistent health check interface, we can leverage cloud-based monitoring tools to detect when there are problems and respond to them appropriately.

If the service discovery agent discovers that there is a problem with a service instance it can take corrective action like shutting down the ailing instance or bringing additional service instances up.

In a microservices environment that leverages REST, the simplest way to build a health check interface is to expose an HTTP end-point that can return a JSON payload and HTTP status code. In a non-Spring boot based microservice, it is often the developer's responsibility to write an endpoint that will return the health of the service.

In Spring Boot exposing an endpoint is trivial and involves nothing more than modifying our maven build file to include the Spring Actuator module. Spring Actuator provides a number out of the box operational endpoints that will help you understand and manage the health of your service. To use Spring Actuator, you need to make sure you include the following dependencies in your maven build file.

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

If you hit the http://localhost:8080/health endpoint on the licensing service, we should see the following returned:

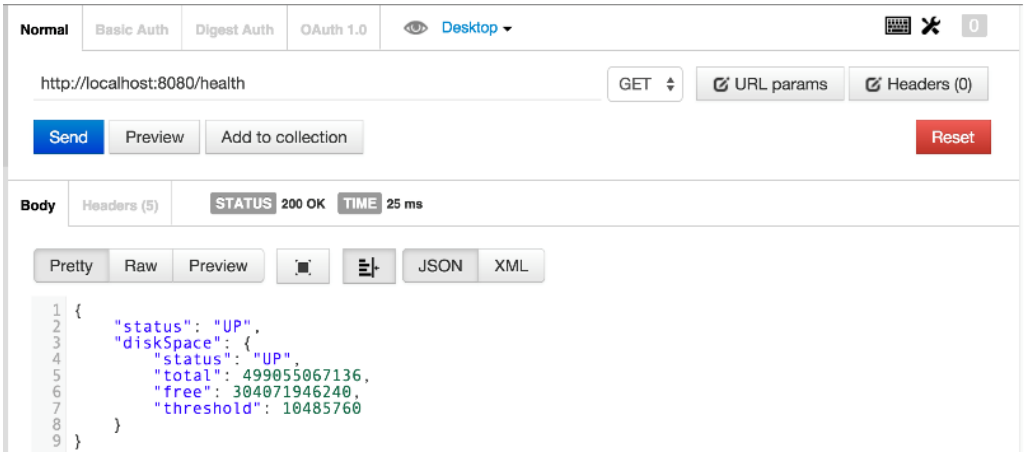

Figure 2.11 The Spring Actuator Health Check

As you can see from the image above the health check can be more than just an indicator of what is up and down. It also can give information about the state of the server the microservice instance is running on. This allows for a much richer monitoring experience.[5]

2.5 Pulling the perspectives together

Microservices in the cloud seem deceptively simple. However, in order to be successful with them, you need to have an integrated view that pulls in the perspective of the architect, the developer and DevOps engineer together into cohesive vision. The key takeaways for each of these perspectives are

1. **Architect.** Focus on the natural contours of your business problem. Describe your business problem domain and listen to the story you are telling. Target microservice candidates will emerge. Remember too that it is better to start with a "coarse-grained" microservice and refactor back to smaller services then start with a large group of small services. Microservice architectures, like most good architectures are emergent and not pre-planned to the minute details.
2. **Software Engineer.** Just because the service is small, does not mean good design principles get thrown out the window. Focus on building a layered service where each layer in the service has discrete responsibilities. Avoid the temptation to build frameworks in your code and try to make each microservice completely independent.

[5] Spring Boot offers a significant number of options for customizing your health check. For more details on this, please check out the excellent book *Spring Boot in Action*. Craig Walls gives an exhaustive overview of all the different mechanisms for configuring Spring Boot Actuators.

Premature framework design and adoption can have massive maintenance costs later in the lifecycle of the application.

3. **DevOps Engineer.** Services do not exist in a vacuum. Establish the lifecycle of your services early. The DevOps perspective needs to focus not only how to automate the building and deployment of a service, but also on how to monitor the health of the service and react when something goes wrong. Operationalizing a service often takes more work and forethought then just writing business logic.

2.6 Summary

- In order to be successful with microservices, you need to integrate in the architect's, engineer's, and DevOps' perspectives.
- Microservices, while a powerful architectural paradigm, have their benefits and tradeoffs. Not all applications should be a microservices application.
- From an architect's perspective, microservices are small, self-contained and distributed that have very narrow boundaries and manage a very small set of data.
- From a developer's perspective, microservices are typically built using a REST-style of design and JSON as the payload for sending and receiving data from the service.
- Spring Boot is ideal framework for building microservices because it lets you build a REST-based, JSON service with just a few simple annotations.
- From a DevOp's perspective, how a microservice is packaged, deployed, and monitored are of critical importance.
- Out of the box, Spring Boot allows you deliver a service as a single executable JAR file. An embedded Tomcat server in the producer JAR file hosts the service.
- Spring Actuator, which is included with the Spring Boot framework exposes information about the operational health of the service along with information about the services runtime.

3

Controlling your configuration in the Cloud with Spring Cloud Config

This chapter covers

- **Why separating service configuration from service code is critical to the deployment of a cloud-based microservice**
- **Configuring a Spring Cloud Configuration Server to serve different configuration profiles using the file system and GIT**
- **Integrating a Spring Boot Microservice with Spring Cloud Config Server**

At one point or another a developer will be forced to separate configuration information from their code. After all it has been drilled into their heads since school that they shouldn't hard-code values into the application code. Some developers will use a constants class file in their application to help centralize all of their configuration in one place. Application configuration data written directly into the code is often problematic because every time a change to the configuration has to be made the application has to be recompiled and/or re-deployed. To avoid this developers, will separate the configuration information from the application code completely. This makes it easy to make changes to configuration without going through a recompile process, but also introduces complexity as you now have another artifact that needs to be managed and deployed with the application.

Many developers will turn to the lowly property file (or YAML, JSON or XML) to store their configuration information. This property file will sit out on a server often containing database and middleware connection information and meta-data about the application that will drive the application's behavior. Segregating your application into a property file is easy and most developers never do any more operationalization of their application configuration then placing

their configuration file under source control (if that) and deploying it as part of their application.

This approach might work with a small number of applications, but it quickly falls apart when dealing with cloud-based applications that might contain hundreds of microservices where each microservice in turn might have multiple service instances running.

Suddenly configuration management becomes a big deal as application and operations team in a cloud-based environment have to wrestle with a rat's nest of what configuration files go where. Cloud based development emphasizes:

1. Completely separating the configuration of an application from the actual code being deployed.
2. Building the server and the application and an immutable image that **never** changes as it is promoted through your environments.
3. Injecting any application configuration information at startup time of the server through either environment variables or through a centralized repository the application's microservices read on startup.

This chapter will introduce you to the core principles and patterns needed to managing application configuration data in a cloud-based application.

3.1 Onmanaging configuration (and complexity)

Managing application configuration is critical for a microservice, cloud-based application, because microservices running in the cloud need to be able to be launched quickly with minimal human intervention. Every time a human being needs to touch manually configure or touch a service to get it deployed is an opportunity for configuration drift and outage.

Let's begin our discussion about application configuration for cloud-based microservices by establishing four principles we want to follow.

1. **Segregate.** We want to completely separate the services configuration information from the actual physical deployment of a service. Application configuration should not be deployed with the service instance. Instead configuration information should either be passed to the starting service as environment variables or read from a centralized repository when the service starts.
2. **Abstract**. Abstract the access of the configuration data behind a service interface. Rather than writing code that directly accesses the service repository (e.g. read the data out of a file or a database using JDBC), have the application leverage a REST-based JSON service to retrieve information.
3. **Centralize.** Since a cloud-based application might literally have hundreds of services it is critical to minimize the number of different repositories used to hold configuration information. Centralize your application configuration into as few repositories as possible.

4. **Harden.** Since your application configuration information is going to be completely segregated from your deployed service and centralized it is critical that whatever solution you utilize can be implemented to be highly available and redundant.

One of the key things to remember is that when you separate your configuration information outside of your actual code you are creating an external dependency that will needs to be managed and version controlled. The author cannot emphasize enough that the application configuration data needs to be tracked and version-controlled as mismanaged application configuration is a fertile ground breeding group for difficult to detect bugs and unplanned outages.

On accidental complexity

The author has experienced firsthand the dangers of not having a strategy for managing your application configuration data. While working at a Fortune 500 financial services company, the author was asked to help bring a large WebSphere upgrade project back on track. The company question had over 120 applications on WebSphere and needed to upgrade their infrastructure from WebSphere 6 to WebSphere 7 before the entire application environment went end of life.

The project had already been going on for a year and had only 1 out of 120 applications deployed. The project had spent a million dollars of effort in people and hardware costs and with their current trajectory was on track to take another two years finishing the upgrade.

When the author started working with the application team one (and just one) of the major problems he uncovered is that the application team managed all of their configuration for their databases and the endpoints for their services inside of property files. These property files were managed by hand and were not under source control. With a 120 applications spread across four environments and multiple WebSphere nodes for each application this literally turned into the team trying to migrate 12,000 configuration files that were spread across all of 100s of servers and applications running on the server. (You are reading this number right: 12,000). These files were just for application configuration and not even application server configuration.

The author convinced the project sponsor to take two months to consolidate all of the application information down to a centralized, version controlled configuration repository with 20 configuration files. When the author asked the framework team how it got to the point where they had 12,000 configuration files, the lead engineer on the team said originally they designed their configuration strategy around a very small group of applications. However, the number of web applications built and deployed exploded in a 5 year period and even though they begged for money and time to rework their configuration management approach their business partners and IT leaders never considered it a priority.

Not spending the time upfront figuring out how you are going to do configuration management can have real (and costly) downstream impacts.

3.1.1 Our configuration management architecture

If you remember from the previous chapter, the loading of configuration management for a microservice occurs during the bootstrapping phase of the microservice.

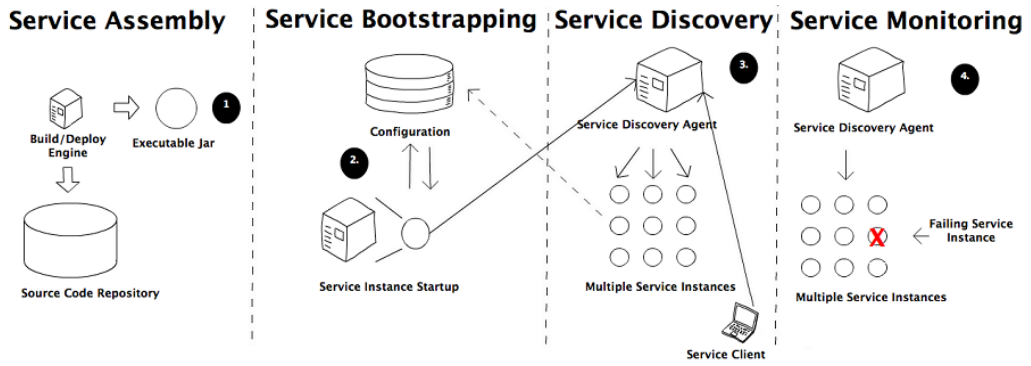

Figure 3.1 Reading of the application configuration data occurs during the service bootstrapping phase

Let's take our four principles we laid out earlier and go through them in more detail. The diagram below shows how the configuration management part of the bootstrapping process will work.

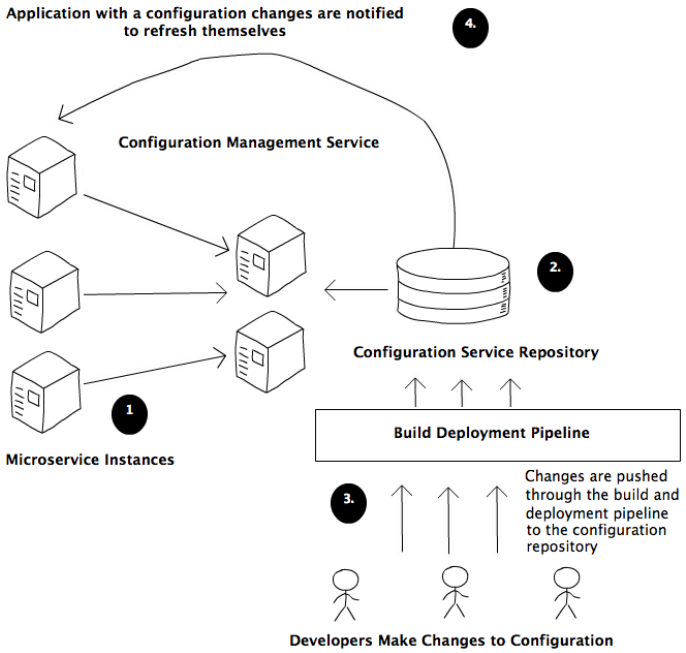

Figure 3.2 Configuration management conceptual architecture

In Figure 3.2, we see several activities taking place:

1. When a microservice instance comes up it is going to call a service endpoint to read its configuration information that is specific to the environment it is operating in. The connection information for the configuration management (connection credentials, service endpoint, and so on) will be passed into the microservice when it starts up.
2. The actual configuration will reside in a repository. Based on the implementation of your configuration repository, you can choose to leverage a number of different implementations to hold your configuration data. The implementation choices can include files under source control, a relational database or a key-value data store.
3. The actual management of the application configuration data occurs independently of how the application is deployed. Changes to configuration management are typically handled through the build and deployment pipeline where changes of the configuration can be tagged with version information and deployed through the different environments.
4. When a configuration management change is made, the services that use that application configuration data must be notified of the change and refresh their copy of the application data.

At this point we have worked through the conceptual architecture that illustrates the different pieces of a configuration management pattern and how these pieces fit together. We are now going to move on to look at the different solutions for the pattern and then at a concrete implementation.

3.1.2 Implementation choices

Fortunately, there are a large number of battle-tested open source projects that can be used to implement a configuration management solution. Let's look at some of the different choices available and compare them. Table 3.1 lays out these choices.

Table 3.1 Open-source projects that can be used to implement a configuration management system.

Project Name	Description	Characteristics
Etcd	Open source project written in Go. Used for service discovery and key-value management. Uses the raft[6] protocol for its distributed computing model.	• Very fast and scalable • Distributable • Command-line driven • Easy to use and setup

[6] https://raft.github.io/

Eureka	Written by Netflix. Is extremely battle tested. Used for both service discovery and key-value management.	• Distribute key-value store. • Very flexible takes some effort to setup to use. • Offers dynamic client refresh out of the box
Consul	Written by Hashicorp. Very similar to Etcd and Eureka in features, but uses a different algorithm for its distributed computing model (SWIM protocol [7])	• Very fast • Offers native service discovery with the option to integrate directly with DNS • Does not offer client dynamic refresh right out of the box
Zookeeper	An Apache project that offers distributed locking capabilities. Is also often used as a configuration management solution for accessing key-value data.	• Oldest most battle tested of the solutions. • Is the most complex to use • Can be used for configuration management but should be considered only if you are already using Zookeeper in other pieces of your architecture
Spring Cloud Configuration Server	SpringSource open source project that offers a general configuration management solution with different back ends. It can integrate with GIT, Eureka and Consul as a backend.	• Non-distributed key/value store • Offers tight integration for Spring and non-spring services • Can use multiple back-ends for storying configuration data including a shared filesystem, Eureka, Consul and Git.

All of the solutions above can easily be leveraged to build a configuration management solution. For our examples in this Chapter and throughout the rest of the book, we are going to use Spring Cloud Configuration server. We chose this solution for several reasons, including:

1. Spring Cloud Configuration Server is easy to use and set up.
2. Spring Cloud Configuration integrates tightly with Spring Boot. We can literally read all of our application's configuration data with a few simple to use annotations.
3. Spring Cloud Configuration Server offers multiple back ends for storing configuration data. If you are already using tools like Eureka and Consul you can plug them right into Spring Cloud Configuration server.
4. Of all of the solutions above, Spring Cloud Configuration server can integrate directly with the Git source control platform. Spring Cloud Configurations integration with Git eliminates an extra dependency in your solutions and makes versioning your application configuration data a snap.

[7] https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf

The other tools (etcd, consul, eureka) do not offer any kind of native versioning and you if you wanted that you would have to build that yourself. If your shop uses Git, the use of Spring Cloud Configuration Server is an attractive option.

For the rest of this chapter we are going to

1. Set up a Spring Cloud Configuration Server and demonstrate two different mechanisms for serving application configuration data: one using the filesystem and another using a Git repository.
2. Continue to build out the licensing service to retrieve some data from a database.
3. Hook the Spring Cloud Configuration Service into our licensing Service to serve up application Configuration data.

   Finally, launch all of the examples in this chapter as a Docker cluster to demonstrate how all of the pieces fit together.

3.2 Building our Spring Cloud Configuration Server

The Spring Cloud Configuration server is a REST-based application that is built on top of Spring Boot. It does not come as a standalone server. Instead you can choose to either embed it as an existing Spring Boot application or start a new Spring Boot project with the server embedded it.

The first thing we need to do is setup a new project directory called confsvr. Inside of the consvr directory we are going to create a new maven file that will be used to pull down the jars necessary to start up on our Spring cloud configuration server. Rather than walk through the entire Maven file, I am going to list the key parts of listing 3.1.

Listing 3.1 Setting up the pom.xml for the Spring Cloud Configuration Server

```
<?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.thoughtmechanix</groupId>
<artifactId>configurationserver</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>Config Server</name>
<description>Config Server demo project</description>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config</artifactId>          (1)
<version>1.0.4.RELEASE</version>
<type>pom</type>
```

```
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-config</artifactId> (1)  (2)
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config-server</artifactId>  (1)  (3)
</dependency>
</dependencies>

<!--Docker build Config  Not Displayed -->
</project>
```

(1) 1,2,3 Tells maven the Spring Cloud Config Dependencies that will be needed

In the Maven file above, we starting out by declaring the Spring Cloud Configuration parent POM that we are going to use. **(1)** This parent POM contains all of the third party libraries and dependencies that are used in the cloud configuration server. Then we declare two more dependencies. The first dependency is the `spring-cloud-start` dependency that is used by all Spring Cloud projects **(2)**. The third dependency is the `spring-cloud-config-server` starter project. This contains the core libraries for the spring-cloud-config-server. **(3)**

We still need to setup one more file to get our core configuration server up and running. This file is our `application.yml` file and is located in the `confsvr/src/main/resources`directory. The `application.yml` file will tell our Spring Cloud Configuration service what port to listen to and also where to locate the back-end that will be serving up the configuration data.

We are almost ready to bring up our Spring Cloud Configuration service. We just need to point the server to a back end repository that will hold our configuration data. For this chapter we are going to use the licensing service that we began to build in Chapter 2 as our example of how to leverage Spring Cloud Config. To keep things simple, we are going to setup application configuration data for three environments: a default environment for when we run the service locally, a dev environment and a production environment.

In Spring Cloud Configuration, everything works off a hierarchy. Your application configuration is represented by the name of the application and then a property file for each environment you want to have configuration information for. In each of these environments we are going to setup two configuration properties:

- An example property that is going to be used by our licensing service directly
- The Database configuration for the Postgres database we are going to use to store our licensing service data.

Figure 3.3 illustrates how we are going to setup and use the Spring Cloud Configuration service. One thing to note is that as we build out with our config service is that it just going to

be another microservice running in our environment. Once it is setup, the contents of the service can be access via a http-based, REST endpoint.

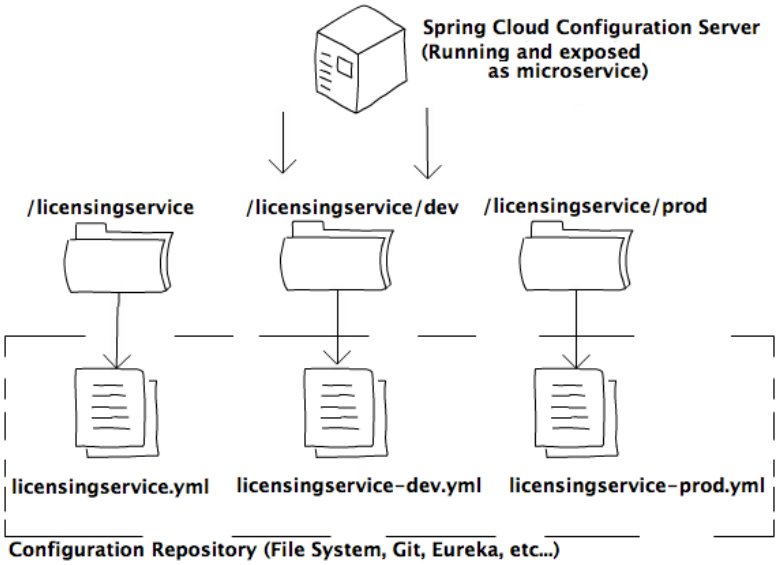

Figure 3.3 Spring Cloud configuration server property layer

The naming convention for the application configuration files are appname-env.yml. As you can see from the diagram above, the environment names translate directly into the URLs that will be accessed to browse configuration information. Later on when we start our licensing microservice example, the environment we want to run the service against is specified by the Spring Boot profile we pass in on the command-line service startup. If a profile is not passed in on the command line, Spring Boot will always default to the configuration data contained in the property file without an environment prefix will be used.

Here is an example of some of the application configuration data we are going to serve up for the licensing service. This is the data that will be contained within the licensingservice.yml file from figure 3.3.

```
tracer.property: "I AM THE DEFAULT"
spring.jpa.database: "POSTGRESQL"
spring.datasource.platform:  "postgres"
spring.jpa.show-sql: "true"
spring.database.driverClassName: "org.postgresql.Driver"
spring.datasource.url: "jdbc:postgresql://192.168.99.100:5432/eagle_eye_local"
spring.datasource.username: "postgres"
spring.datasource.password: "p0stgr@s"
spring.datasource.testWhileIdle: "true"
spring.datasource.validationQuery: "SELECT 1"
spring.jpa.properties.hibernate.dialect: "org.hibernate.dialect.PostgreSQLDialect"
```

Let's now go set up our Spring Cloud Configuration Server with our simplest example, the filesystem.

Think before you implement

The authors recommends against using a filesystem-based solution for a medium to large cloud applications. Using the filesystem approach means that you need to implement a shared file mount point for all Cloud Configuration servers that want to access the application configuration data. Setting up share filesystem servers in the cloud are doable, but it does force the onus of maintaining this environment back on you.

The author is showing the filesystem approach is the easiest example use when getting your feet wet with Spring Cloud Configuration server. In a later section, we will demonstrate how to configure Spring Cloud Configuration server to use a cloud-based Git provider like BitBucket or GitHub to store your application configuration.

3.2.1 Using Spring Cloud Configuration Server with the filesystem

The Spring Cloud Configuration server uses an entry in the `confsvr/src/main/resources/application.yml` file to point to the repository that will hold the application configuration data. Setting up a filesystem-based repository is the easiest way to accomplish this.

To do this, add the following information to the configuration servers `application.yml` file.

Listing 3.2 Spring Cloud Configuration's application.yml file

```
server:
   port: 8888          ❶
spring:
  profiles:
    active: native     ❷
  cloud:
     config:
       server:
           native:
              searchLocations: file:///Users/johncarnell1/book/native_cloud_apps/ch4-config-
      managment/confsvr/src/main/resources/config/licensingservice  ❸
```

❶ Port the Spring Cloud Configuration Server will listence on
❷ The backend repository (filesystem) that will be used to store the configuration
❸ The path to where the configuration files are stored

In the configuration file above we started by telling the configuration server what port number it should listen to for all requests for configuration.

```
server:
   port: 8888
```

Since we are using the file system for storing application configuration information, we need to tell Spring Cloud configuration server to run with the "native" profile.

```
profiles:
   active: native
```

The last piece in our `application.yml` file above provides Spring Cloud Configuration with the directory in which the application data resides.

```
server:
   native:
      searchLocations: file:///Users/johncarnell1/book/native_cloud_apps/ch4-config-
       managment/confsvr/src/main/resources/config/licensingservice
```

The important parameter in the configuration entry above is the `searchLocations` attribute. This attribute provides a comma separated list of the directories for each application that is going to have properties managed by the configuration server. In the example above we only have the licensing service configured.

We now have enough work done to start the configuration server. Let's go ahead and start the configuration server using the `mvn spring-boot:run` command. The server should now come up with the Spring Boot splash screen on the command line. If we point our browser over to http://localhost:8888/licensingservice/default we will see JSON payload being returned with all of properties contained within the `licensingservice.yml` file.

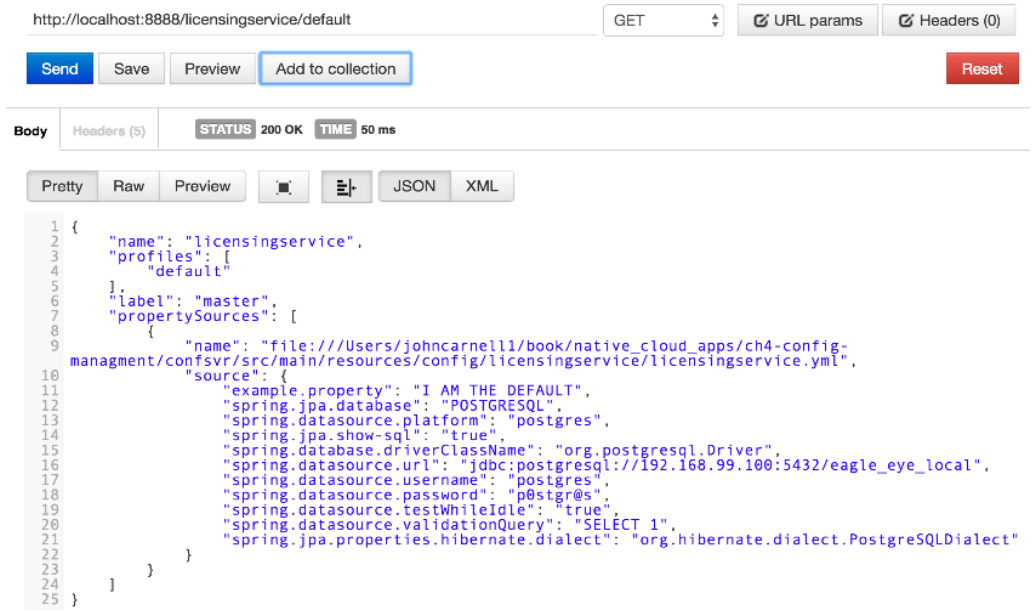

Figure 3.4 Retrieving default configuration information for the licensing service

If you wanted to see the configuration information for the dev licensing service environment hit the http://localhost:8888/licensingservice/dev endpoint.

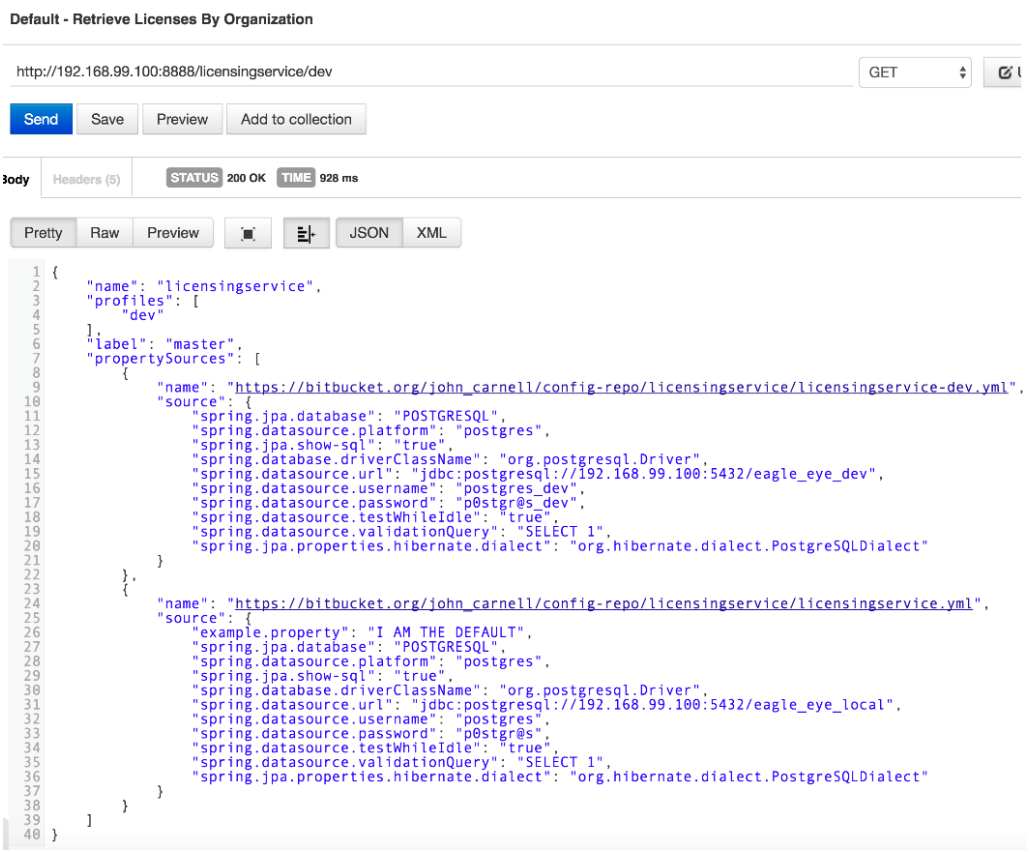

Figure 3.5 Retrieving Configuration Information Using the Dev Profile

If you look closely, when you hit the dev endpoint you are returning back both the default configuration properties for the licensing service and also the dev licensing service configuration. The reason why Spring Cloud Configuration is returning both sets of configuration information is that the Spring framework implements a hierarchical mechanism for resolving properties. When the Spring Framework does property resolution, it will always look for the property in the default properties first and then override the default with an environment specific value if one is present.

In concrete terms, if you define a property in the `licensingservice.yml` file and do not define it in any of the other environment configuration files (e.g. the `licensingservice-dev.yml`), the Spring framework will use the default value.

Note: This is not the behavior you will see by directly calling the Spring Cloud Configuration REST endpoint. The REST endpoint will return all configuration values for both the default and environment specific value that was called.

On Spring Cloud configuration and security

An astute reader will notice that the database credentials for the different environments have been exposed as clear text via the REST endpoint. Spring Cloud Configuration Server offers the ability to encrypt sensitive property files. We are not going to be walking through how to encrypt the property files in this chapter or how to secure the Spring Cloud Configuration server using different authentication and authorization mechanisms.

The subject of security is extensive and covered in the Spring Cloud Configuration documentation at http://cloud.spring.io/spring-cloud-config/spring-cloud-config.html#_spring_cloud_config_server.

So let's go ahead and look now at how can hook up the Spring Cloud Configuration server to our licensing microservice.

3.3 Integrating Spring Cloud Configuration with a Spring Boot Client

In the previous chapter, we built a simple skeleton of our licensing service that did nothing more than returns a hardcoded Java object representing a single licensing record from our database. In this next example, we are going to build out the licensing service and talk to a Postgres database holding our licensing data.

We are going to communicate with the database using Spring Data and map our data from the licensing table to a POJO holding our data. Our database connection and a simple property are going to be read out of Spring Cloud Configuration server. Figure 3.6 shows what is going to happen between our licensing service and the Spring Cloud Configuration Service.

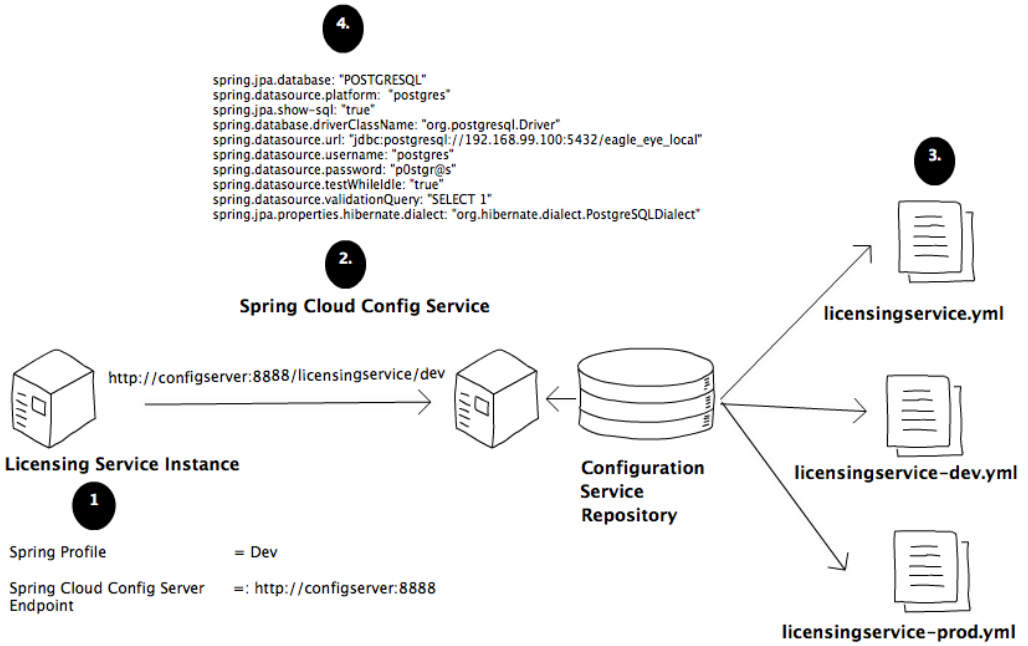

Figure 3.6 Retrieving Configuration Information Using the Dev Profile

When our licensing service is first started we are going to pass it via the command-line two pieces of information: The Spring profile and the endpoint the licensing service should use to communicate with the Spring Cloud Configuration service. **(1)**The Spring profile value maps to the environment of the properties being retrieved for the Spring service. When the licensing service first boots up it will contact the Spring Cloud Config service via an endpoint built from the Spring profile passed into it **(2)**. The Spring Cloud Config service will then use the configured back end config repository (filesystem, Git, Consul, Eureka) **(3)** to retrieve the configuration information specific to the Spring profile value passed in on the URI. The appropriate property values **(4)** are then passed back to the licensing service. The Spring Boot framework will then inject these values into the appropriate parts of the application.

3.3.1 Setting up the Licensing Service Spring Cloud Config Server Dependencies

So let's change our focus from the configuration server to the licensing service. The first thing we need to do is add a couple of more entries to our maven file in our licensing service. The entries that need to be added include are shown in listing 1.2 below:

Listing 3.3 Additional Maven Dependencies needed by the Licensing Service

```
<dependency>
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-data-jpa</artifactId>      (1)
</dependency>

<dependency>
<groupId>postgresql</groupId>
<artifactId>postgresql</artifactId>                        (2)
<version>9.1-901.jdbc4</version>
</dependency>

<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config-client</artifactId>        (3)
<version>1.0.4.RELEASE</version>
</dependency>
```

(1) 1 – Tells Spring Boot we are going to use Java Persistence API (JPA) in our service
(2) 2 – Tells Spring Boot to pull down the Postgres JDBC drivers
(3) 3 – Tells Spring Boot that we should pull down the dependencies need for the Spring Cloud Config

The first and second dependencies, `spring-boot-starter-data-jpa` and postgresql, import the Spring Data Java Persistence API (JPA) **(1)** and the Postgres JDBC drivers**. (2)** The last dependency, the `spring-cloud-config-client` contains all of the class needed to interact with the Spring Cloud Configuration Server. **(3)**

3.3.2 Configuring the Licensing Service to use Spring Cloud Config

After the maven dependencies have been defined, we now need to tell the licensing service where to contact that Spring Cloud Configuration server. This is done by setting two properties in the licensing services `licensing-service/src/main/resources/application.yml` file.

Listing 3.4 Configuring the licenseservices application.yml

```
spring:
 application:
    name: licensingservice    (1)
  profiles:
    active:
      default                 (2)
  cloud:
    config:
      uri: http://localhost:8888    (3)
```

(1) 1 – Specify the name of the licensing service so that Spring Cloud Config client knows which service is being looked
(2) 2 – Specify the default profile the service should run. Profile maps to environment
(3) 3 – Specify the location of the Spring Cloud Config server.

These properties are the `spring.application.name`, the `spring.profiles.active` and the `spring.cloud.config.uri`. Please note that the Spring Boot applications support multiple mechanisms to define a property. We chose YAML (Yet Another Markup Language) as the

means for configuring our application. The hierarchical format of the property values map directly to the `spring.application.name`, `spring.profiles.active`, `spring.cloud.config.uri` names.

The`spring.application.name`**(1)** is the name of our application (e.g. licensingservice) and **must** map directly to the name of the directory within our Spring Cloud Configuration server. So for the licensingservice, we want a directory on the Spring Cloud Configuration server called licensing.

The second property, the `spring.profiles.active`**(2)** is used to tell Spring Boot what profile the application should run as. A profile is a mechanism to differentiate the configuration data consumed by the Spring Boot application. For the licensingservice's profile, the environment the service is going to map directly to the environment we are going to support in our cloud configuration environment. So for instance by passing in dev as our profile, the Spring Cloud Config server will use the dev properties. If you set a profile, the licensing service will use the default profile.

The third and last property, the `spring.cloud.config.uri`**(3)** is the location of where the licensing service should look for the Spring Cloud Configuration server endpoint. By default, the licensing service will look for the configuration server at http://localhost:8888. Later on in

the chapter we will demonstrate how to override the different properties defined in the `application.yml` file on application startup. This will allow us to tell the licensing microservice which environment it should be running in.

Now if we bring up the Spring Cloud Configuration Services and the corresponding database all running on your local machine we can launch the licensing service using its default profile.

```
bash-3.2$ mvn spring-boot:run
```

Figure 3.7 Launching the Default Licensing Service

By running this command without any properties set, the licensing server will automatically

attempt to connect to the Spring Cloud Configuration server using the endpoint (http://localhost:8888) and the `active` profile (default) defined in the `application.yml` file of the licensing service.

If you want to override these default values and point to another environment, you can do this by compiling the licensingservice project down to a jar and then run the jar with a -D system property override. The example below demonstrates how to launch Spring Cloud Configuration Server locally and override it to use the "dev" profile.

```
bash-3.2$ java -Dspring.cloud.config.uri=http://192.168.99.100:8888
-Dspring.profiles.active=dev -jar target/licensing-service-0.0.1-SNAPSHOT.jar
```

Figure 3.8 Overriding the default behavior on Spring Cloud Configuration Server

With the command line above we are overriding the two parameters: `spring.cloud.config.uriand spring.profiles.active`. With the `-Dspring.cloud.config.uri=http://192.168.99.100:8888` system property we are pointing to a configuration server running away from our local box. With the `-D spring.profiles.active=dev` system property we are telling the licensing service to use the dev profile (read from the configuration server) to connect to a dev instance of a database.

Use environment variables to pass startup information

In our examples we are hard-coding the values to pass in to the –D parameter values. In the cloud, most of the application config data you need will be in your configuration server. However, for the information you need to start your service (like the data for the configuration server) you would start the VM instance or Docker container and pass in an environment variable. All of the code examples for each chapter can be completely run from within Docker containers. With Docker we simulate different environments through environment specific Docker-compose files that orchestrate the startup of all of our services. Environment specific values needed by the containers are passed in as environment variables to the container. For example, to start our licensing service in a dev environment, the `docker/dev/docker-compose.yml` file contains the following entry for the licensing-service:

```
  licensingservice:

    image: ch3-thoughtmechanix/licensing-service
    ports:
      - "8080:8080"
    links:
      - "database:database"
      - "configserver:configserver"
    environment:  (1)
      PROFILE: "dev"      (2)
      CONFIGSERVER_URI: http://configserver:8888(3)
```

The environment **(1)** entry in the file contains the values of two variables PROFILE **(2)**, which is the Spring Boot profile the licensing service is going to run under. The CONFIGSERVER_URI **(3)** is used passed to our licensing service to our Spring Cloud Configuration server instance.

In our startup scripts that are run by the container we then pass these environment variables as –D parameters to our JVMS starting the application. In each project we bake a Docker container and that Docker container uses a startup script that starts the software in the container. For the licensing service, the startup script that gets baked into the container can be found at licensing-service/src/main/docker/run.sh. In the run.sh script, the following entry starts our licensingservice JVM.

```
echo "********************************************************"
echo "Starting License Server with Configuration Service :  $CONFIGSERVER_URI";
echo "********************************************************"
java -Dspring.cloud.config.uri=$CONFIGSERVER_URI
-Dspring.profiles.active=$PROFILE -jar /usr/local/licensingservice/licensing-service-0.0.1-
SNAPSHOT.jar
```

Since we configure all of our services with the Spring Boot Actuator we can confirm the environment we are running against by hitting http://localhost:8080/env. The `/env` endpoint will provide a complete list of the configuration information about the service including the properties and endpoints the service has booted with.

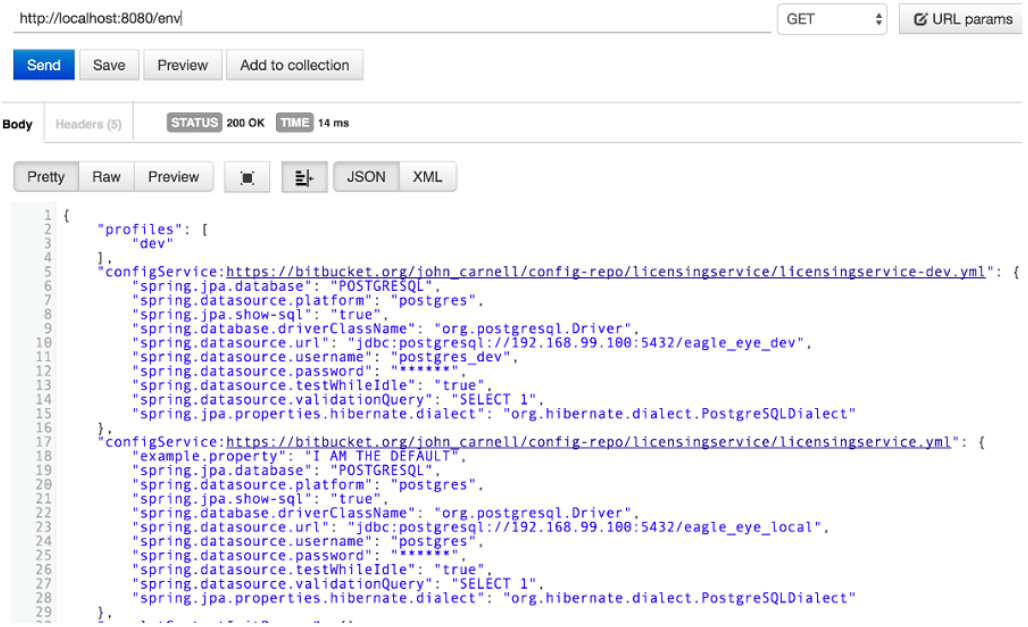

Figure 3.9Checking the environment

The key thing to note from the diagram above is that the active profile for the licensing service is `dev`. By inspecting the returned JSON, you can also see that the Postgres database being returned is a development URI of `jdbc:postgresql://192.168.99.100:5432/eagle_eye_dev`.

On exposing too much information

Every organization is going to have different rules about how to implement security around their services. Many organizations believe services should not broadcast any information about themselves and will not allow things like a /env endpoint to be active on a service as they believe (rightfully so) that this will provide too much information for a potential hacker. Spring Boot provides a wealth of capabilities on how to configure what information is returned by the Spring Actuators endpoints that are the outside the scope of this book. Craig Walls excellent book, Spring Boot In Action covers this subject in detail and I would highly recommend that you review your corporate security policies and Craig's book to provide the right level of detail you want exposed through Spring Actuator.

3.3.3 Wiring in a data source using Spring Cloud Configuration Server

At this point we have the database configuration information being directly injected into our microservice. With the database configuration set, configuring our licensing microservice to becomes an exercise in using standard Spring components to build and retrieve the data from our Postgres database. The licensing service has been refactored into different classes with each class having separate responsibilities. These classes are shown in table 3.2.

Table 3.2 Licensing service classes and locations

Class Name	Location
License.java	licensing-service/src/main/java/com/thoughtmechanix/licenses/model
LicenseRepository	licensing-service/src/main/java/com/thoughtmechanix/licenses/repository
LicenseService	licensing-service/src/main/java/com/thoughtmechanix/licenses/service

The License class is our model class that will hold the data retrieved from our licensing database.

Listing 3.5 The JPA Model code for a single license record

```
package com.thoughtmechanix.licenses.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity                                              (1)
@Table(name = "licenses")                            (2)
public class License{
  @Id                                                (3)
  @Column(name = "license_id", nullable = false)     (4)
  private String licenseId;

  @Column(name = "organization_id", nullable = false)
  private String organizationId;

  @Column(name = "product_name", nullable = false)
  private String productName;

  /*The rest of the code has been removed for conciseness*/

}
```

(1) 1 - @Entity tells Spring that this is a JPA class
(2) 2 - @Table maps to the database table
(3) 3 - @Id marks this field as a primary key
(4) 4 - @Column maps the field to a specific database table

The class uses a number of Java Persistence Annotations (JPA) that help the Spring Data framework map the data from the licensing table in the Postgres database to the Java object. The `@Entity` annotation **(1)** lets Spring know that this Java POJO is going to be mapping object that will hold data. The `@Table` annotation **(2)** tells Spring/JPA what database table should be mapped. The `@Id` annotation identifies the primary key for the database. **(3).** Finally, each one of the columns from the database that are going to be mapped to individual properties is marked with a `@column`**(4)** attribute.

The Spring Data and JPA framework provides your basic CRUD (Create Read Update and Delete) methods for accessing a database. If you want to build methods beyond that you can use a Spring Data Repository interface and some basic naming conventions to build those methods. Spring will at startup parse the name of the methods from the Repository interface, convert them over to a SQL statement based on the names and then generate a dynamic proxy class under the covers to do the work. The repository for the licensing service, is shown below:

Listing 3.6 LicenseRepository Interface defines the query methods

```
package com.thoughtmechanix.licenses.repository;

import com.thoughtmechanix.licenses.model.License;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository  (1)
public interface LicenseRepository extends CrudRepository<License,String>  (2)  {
    public List<License> findByOrganizationId(String organizationId);       (3)
    public License findByOrganizationIdAndLicenseId(String organizationId,String licenseId);
}
```

(1) 1 – Tells Spring Boot that this is a JPA repository class
(2) 2 – Defines that we are extending the Spring CrudRepository
(3) 3 – Individual query methods are parsed by Spring into a SELECT..FROM query

Our repository interface, `LicenseRepository`, is marked with the `@Repository` annotation **(1)** which tells Spring that it should treat this interface as a repository andgenerate a dynamic proxy for it. Spring offers different types of repositories for data access. We have chosen to use the Crud Repository. **(2)** Finally, we have added two custom methods for query methods for retrieving data from the licensing table. **(3)** The Spring Data framework will pull apart the name of the methods to build a query to access the underlying data.

NOTE The Spring Data framework provides an abstraction layer over various database platforms and is not just limited to relational databases. NoSQL databases like MongoDB and Cassandra are also supported.

Unlike the previous incarnation of the licensing service in Chapter 2, we have now separated the business and data access logic for the licensing service out of the `LicenseController` and into a standalone Service class called `LicenseService`.

Listing 3.7 LicenseService class used to execute database commands

```
package com.thoughtmechanix.licenses.services;

import com.thoughtmechanix.licenses.config.ServiceConfig;
import com.thoughtmechanix.licenses.model.License;
import com.thoughtmechanix.licenses.repository.LicenseRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.UUID;

@Service
public class LicenseService {

    @Autowired
    private LicenseRepository licenseRepository;

    @Autowired
    ServiceConfig config;

    public License getLicense(String organizationId,String licenseId) {
        License license = licenseRepository.findByOrganizationIdAndLicenseId(
organizationId, licenseId);
        return license.withComment(config.getExampleProperty());
    }

    public List<License> getLicensesByOrg(String organizationId){
        return licenseRepository.findByOrganizationId( organizationId );
    }

    public void saveLicense(License license){
        license.withId( UUID.randomUUID().toString());
        licenseRepository.save(license);
    }
            /*Rest of the code removed for conciseness*/
}
```

The controller, service and repository classes are wired together using the standard Spring `@Autowired`annotation.

3.3.4 Directly Reading Properties using the @Value Annotation

In the `LicenseService` class above the, you might have noticed that we are setting the `license.withComment()` value in the `getLicense()` code with a value from the `config.getExampleProperty()` class. The code being referred to is below.

```
public License getLicense(String organizationId,String licenseId) {
        License license = licenseRepository.findByOrganizationIdAndLicenseId(
organizationId, licenseId);
return license.withComment(config.getExampleProperty());
    }
```

If you look at the `licensing-service/src/main/java/com/thoughtmechanix/licenses/config/ServiceConfig.java` class, you will see property annotated with the `@Value` annotation.

Listing 3.8 ServiceConfig is used to centralize application properties

```
package com.thoughtmechanix.licenses.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class ServiceConfig{

  @Value("${example.property}")
  private String exampleProperty;

  public String getExampleProperty(){
    return exampleProperty;
  }
}
```

While, Spring Data "auto-magically" injects the configuration data for the database into a database connection object, all other properties must be injected using the `@Value` annotation. With the example above, the @Value annotation pulls `the example.property` from the Spring Cloud Configuration server and injects it into the `example.property` attribute on the `ServiceConfig` class.

> **TIP** While it is possible to directly inject configuration values into properties in individual classes, the author has found it useful to centralize all of the configuration information into a single configuration class and then inject the configuration classinto where it is needed.

3.3.5 Using Spring Cloud Configuration Server with Git

As we mentioned earlier, using a filesystem as the backend repository for Spring Cloud Configuration server can be impractical for a cloud-based application because the development

team has to setup and manage a shared file system that is mounted on all instances of the cloud configuration server.

Spring Cloud configuration server integrates with a number of different backend repositories that can be used to host application configuration properties. One the author has used successfully is to leverage Spring Cloud Configuration server with a Git source control repository.

By leveraging Git you can get all of the benefits of putting your configuration management properties under source control and provide an easy mechanism to integrate the deployment of your property configuration files in your build and deployment pipeline.

To leverage Git we would swap out the filesystem back configuration in the configuration service's `application.yml` file with the following configuration:

Listing 3.9 Spring Cloud Config application.yml

```
server:
  port: 8888
spring:
  cloud:
    config:
      server:
        git:                                                    (1)
          uri: https://github.com/carnellj/config-repo/         (2)
          searchPaths: licensingservice,organizationservice     (3)
          username: native-cloud-apps
          password: 0ffended
```

(1) 1 – Tells Spring Cloud Config to use Git as a backend repository
(2) 2 - Tells Spring Cloud Config the URL to the Git server and Git repo
(3) 3 - Tells Spring Cloud Config what the path in Git is to look for config files

The three key pieces of configuration in the example above are the `spring.cloud.config.server`, `spring.cloud.config.server.git.uri` and the `spring.cloud.config.server.git.searchPaths` properties. The `spring.cloud.config.server`**(1)** property tells Spring Cloud Configuration server to use a non-filesystem based backend repository. In our example above we are going to connect to the cloud-based Git repository, Github.

The spring.cloud.config.server.git **(2)** properties provide the URL of the repository we are connecting to. Finally, the `spring.cloud.config.server.git.searchPaths`**(3)** property tells Cloud Config server the relative paths on the Git repository that should be searched when the Cloud Configuration server comes up. Like the filesystem version of the configuration, the value in `seachPaths` will be a comma separated list for each service hosted by the configuration service.

3.3.6 Refreshing your properties using Spring Cloud Configuration Server

One of the first questions that comes up from development teams when they want to use Spring Cloud Configuration server is how can they dynamically refresh their applications when

a property changes. The Spring Cloud Configuration server will always serve the latest version of a property. So changes made to a property via its underlying repository will be up to date.

However, Spring Boot applications will only read their properties at startup time. This means property changes made in the Spring Cloud Configuration server will not be automatically picked up by the Spring Boot application. However, Spring Boot Actuator does offer a `@RefreshScope` annotation that will allow a development team the access a `/refresh` endpoint that will force the Spring Boot application to re-read its application configuration. The code below shows the `@RefreshScope` annotation in action.

Listing 3.10 The @RefreshScope annotation

```
package com.thoughtmechanix.licenses;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.context.config.annotation.RefreshScope;


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

There are a couple of things to note about the ***@RefreshScope*** annotation. First, the annotation will only re-load the custom Spring properties you have in your application configuration. Items like your database configuration that are used by Spring Data will not be reloaded by the @RefreshScope annotation. To actually perform the refresh, you can hit the `http://<yourserver>:8080/refresh` endpoint.

On Refreshing Microservices

When using Spring Cloud Configuration Service with microservices, one thing you need to consider before you dynamically change properties is that you might have multiple instances of the same service running and you will need to refresh all of those services with their new application configurations. There are a number of ways you can approach this problem:

1) Spring Cloud Configuration Service does offer a "push" based mechanism called Spring Cloud Stream that will allow the Spring Cloud Configuration server to publish to all the clients using the service that a change has occurred. Spring Cloud Configuration requires an extra piece of middleware running (rabbitmq). This is an extremely useful means of detecting changes, but not all Spring Cloud Configuration back ends support the "push" mechanism. (e.g. the Consul server)
2) In the next chapter we are going to use Spring Service Discovery and Eureka to register all instances of a service. One technique the author has used to handle application configuration refresh events is to refresh the application properties in Spring Cloud Configuration and then write a simple script to query the service discovery engine to find all instances of a service and call the /refresh endpoint directly.

3) **Finally, you can just restart all of the servers or containers to pick up the new property. This is a trivial exercise especially if you are running your services in a container service like Docker. Restarting Docker containers literally take seconds and this will force a re-read of the application configuration.**

Remember, cloud-based servers are ephemeral. Do not be afraid to start new instances of a service with their new configuration, direct traffic to the new services and then tear down the old ones.

3.4 Summary

Application configuration management might seem like a mundane topic, but it is of critical importance in a cloud-based environment. As we will discuss in more detail in later chapters it is critical that your applications and the servers they run on be immutable and the entire server be promoted between environments. This flies in the face of traditional deployment models where you deployed an application artifact (for example, a jar or war file) along with its property files to "fixed" environment.

With a cloud-based model, the application configuration data should be segregated completely from the application, with the appropriate configuration data needs injected at run-time so that the same server/application artifact are consistently promoted through all environments.

In this chapter, we examined how to

- Set up a Spring Cloud Configuration server with environment specific properties and then launch the application JVM with those environment specific properties.
- Used Spring Profiles to tell Spring Configuration Server which environment the microservice reading its configuration will run under.
- Configure a file based and Git based application configuration repository using Spring Cloud Configuration.
- Integrate Spring Cloud Configuration server with our licensing microservice and used Spring Cloud Configuration server to hold the services custom and database configuration.