Spring Microservices IN ACTION

John Carnell

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Spring Microservices in Action
Version 3

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 for our customers. One example of this churn is the rapid adoption by many organizations of adopting and building applications using microservices. Microservices are distributed, loosely coupled software services that are carry out a very small number of well-defined tasks.

This book introduces 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.

1.1 What is a microservice?

The concept of a microservice originally crept into the software development community's consciousness around 2014. As stated earlier a microservice is very small, loosely coupled, distributed service. Microservices allow you to take a large application and decompose it very easy to manage components that have very specific functionality. Microservices help combat the traditional problems of complexity in a large code based by decomposing the large code based down into very small, well-defined pieces. A microservice architecture has the following characteristics.

- Application logic is broken down into very small-grained components with well-defined boundaries of responsibility that coordinate 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 (Javascript Object Notation) 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 are 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.

1.2 What is Spring and why is 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 normal Java application, the application is decomposed into classes where each class often has explicit linkages to other classes in the application. This is problematic in a large project because these external linkages are brittle and making a change can result in multiple downstream changes. A dependency injection framework allows you to more easily manage large Java projects by externalizing the relationship between objects within your application through convention (and annotations) rather than 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 as bloatware 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 its 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 application's 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] oriented 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 cover REST in more detail in chapter 2, the core concept behind REST is that your services and should 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 oriented 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 wraps several popular cloud management microservice frameworks

[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). It's still one of the best explanations of REST available.

under a common framework and makes the use and deployment of these technologies as easy to use as just annotating your code. I cover the different components within Spring Cloud later on in this chapter.

1.3 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 cover, with hands-on examples,

- What a microservice is and the design considerations that go into building a microservice-based application
- When you shouldn't build a microservice-based application
- How to build microservices using the Spring Boot
- 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 what we have learned and build a deployment pipeline that can be used to deploy to multiple cloud providers

1.4 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 can 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 in seeing 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.

Let's shift gears for a moment and walkthrough building a simple microservice using Spring Boot.

1.5 Building a microservice with Spring Boot

I've 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.

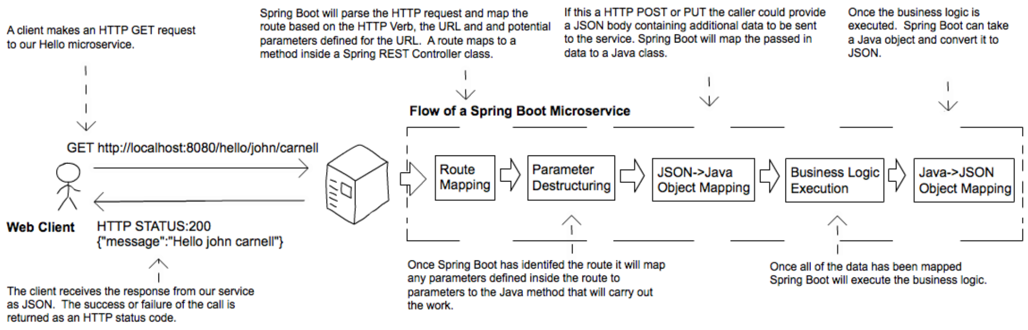

A client makes an HTTP GET request to our Hello microservice.
Spring Boot will parse the HTTP request and map the route based on the HTTP Verb, the URL and and potential parameters defined for the URL. A route maps to a method inside a Spring REST Controller class.
If this a HTTP POST or PUT the caller could provide a JSON body containing additional data to be sent to the service. Spring Boot will map the passed in data to a Java class.
Once the business logic is executed. Spring Boot can take a Java object and convert it to JSON.
Flow of a Spring Boot Microservice
GET http://localhost:8080/hello/john/carnell
Web Client
HTTP STATUS:200
{"message":"Hello john carnell"}
Route Mapping
Parameter Destructuring
JSON->Java Object Mapping
Business Logic Execution
Java->JSON Object Mapping
The client receives the response from our service as JSON. The success or failure of the call is returned as an HTTP status code.
Once Spring Boot has identifed the route it will map any parameters defined inside the route to parameters to the Java method that will carry out the work.
Once all of the data has been mapped Spring Boot will execute the business logic.

Figure 1.1 Spring Boot greatly simplifies the work we have to do to process a user's call into 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 it 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                          ❶
@RestController                                 ❷
@RequestMapping(value="hello")                  ❸
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 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 I'll show shortly) the return of the call would be:

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

So let's fire up our service. To do this, we'll go to the command prompt and issue the following command:

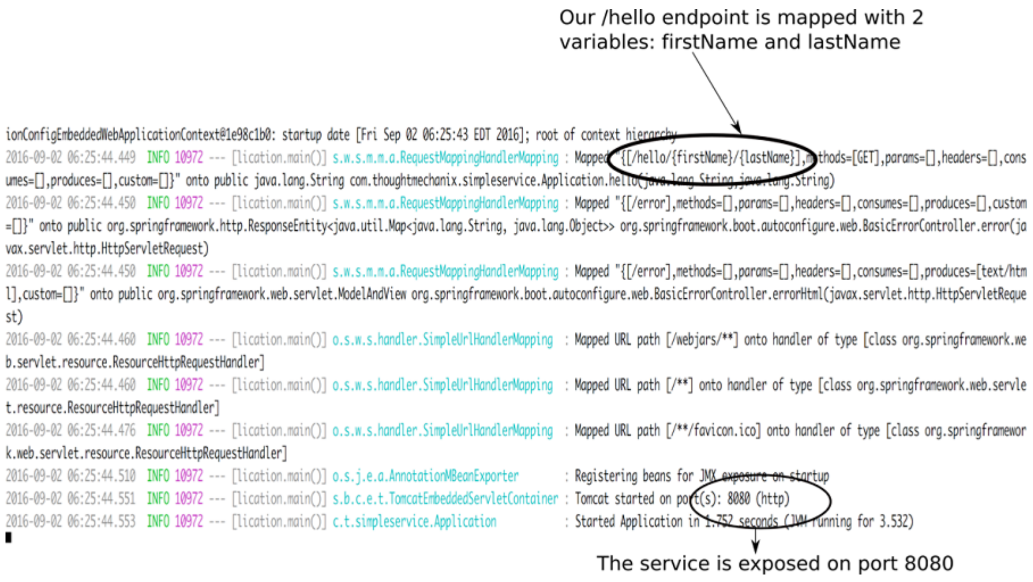
```
mvn spring-boot:run
```

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 limited the examples in this book to Java and Maven. As a long-time Groovy and Gradle aficionado, I have a healthy respect for the language and the build tool, but in an effort to keep the book manageable and the material focused, I have chosen to go with Java and Maven to reach the largest audience.

If everything starts correctly, you should see what's shown in figure 1.2 from your command-line window.


Our /hello endpoint is mapped with 2 variables: firstName and lastName
The service is exposed on port 8080

```
ionConfigEmbeddedWebApplicationContext@1e98c1b0: startup date [Fri Sep 02 06:25:43 EDT 2016]; root of context hierarchy
2016-09-02 06:25:44.449  INFO 10972 --- [lication.main()] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/hello/{firstName}/{lastName}],methods=[GET],params=[],headers=[],cons
umes=[],produces=[],custom=[]}" onto public java.lang.String com.thoughtmechanix.simpleservice.Application.hello(java.lang.String,java.lang.String)
2016-09-02 06:25:44.450  INFO 10972 --- [lication.main()] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[],custom
=[]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicErrorController.error(ja
vax.servlet.http.HttpServletRequest)
2016-09-02 06:25:44.450  INFO 10972 --- [lication.main()] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[text/htm
l],custom=[]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServletReque
st)
2016-09-02 06:25:44.460  INFO 10972 --- [lication.main()] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.we
b.servlet.resource.ResourceHttpRequestHandler]
2016-09-02 06:25:44.460  INFO 10972 --- [lication.main()] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframework.web.servle
t.resource.ResourceHttpRequestHandler]
2016-09-02 06:25:44.476  INFO 10972 --- [lication.main()] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframewor
k.web.servlet.resource.ResourceHttpRequestHandler]
2016-09-02 06:25:44.510  INFO 10972 --- [lication.main()] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2016-09-02 06:25:44.551  INFO 10972 --- [lication.main()] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2016-09-02 06:25:44.553  INFO 10972 --- [lication.main()] c.t.simpleservice.Application            : Started Application in 1.752 seconds (JVM running for 3.532)
```

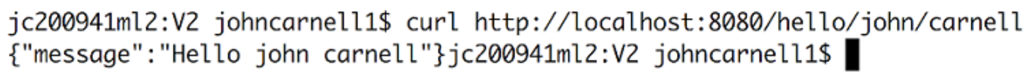
Figure 1.2 Our Spring Boot Service will communicate the endpoints exposed and the port of the service

If you look closely 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. Curl is command-line that lets you issue HTTP requests to an HTTP endpoint. 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 The response from the /hello endpoint shows the data we have requested represented as a JSON payload

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.

1.6 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 databases 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 expect the 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

[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

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 services means 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 service scan easily be distributed horizontally across multiple servers, making it possible 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

1.7 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 (IaaS)
- Platform as a Service (PaaS)
- Software as a Service (SaaS)

To better understand these concepts, let's map the everyday task of making a meal to the 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 get a meal delivered to our houses
4. We can get in the car and eat at restaurant.

Figure 1.4 shows each model.

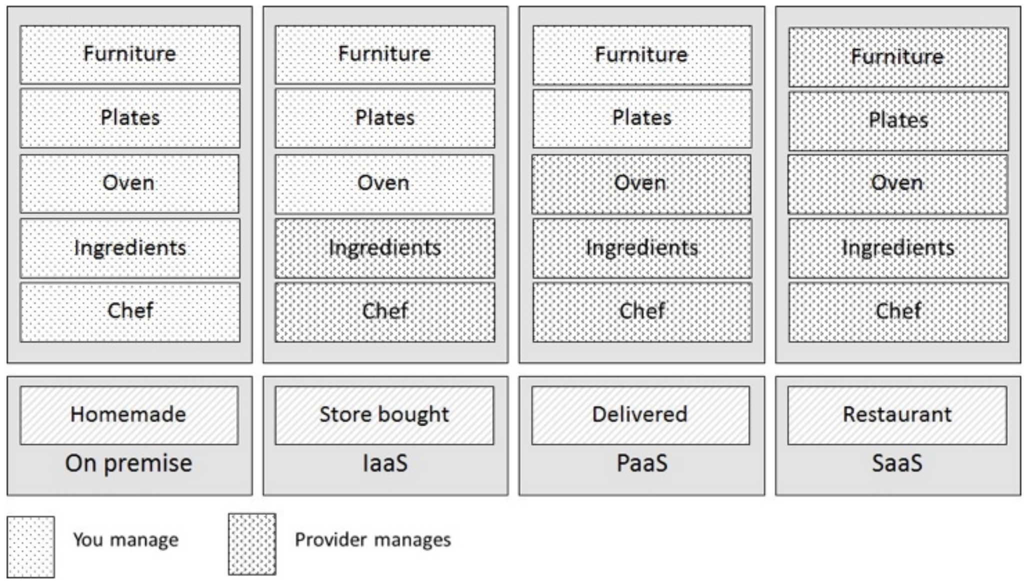


Figure 1.4 The different cloud computing models really come down to who is responsible for what: the cloud vendor or you

The difference between these options is really about who's 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 are still responsible for heating the meal and eating it at the house (and cleaning up the dishes afterwards).

In a Platform as a Service (PaaS) model we still have some responsibilities for the meal, but we further rely on a vendor to take care of the core tasks associated with making a meal. For example in a PaaS model, you have to supply the plates and furniture, but the restaurant owner provides the oven, ingredients and the Chef to cook them. 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 or wash.

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 the application? In a IaaS model, the cloud vendor provides the basic infrastructure, but you are accountable for selecting the technology and building the final solution. On the other end of the spectrum, with a SaaS model, you are a passive consumer of the service provided by the vendor and have no input on the technology selection or any accountability to maintain the infrastructure for the application.

1.8 Why the cloud and microservices?

One of the core concepts of a microservice-based architecture is that each service is packaged and deployed as its own discrete and independent artifact. Services instances 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't quickly ramp up the capacity of a physical server and it can become extremely costly to scale your microservice horizontally across multiple physical servers.
- **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 containers are a natural extension to deploying your microservices on a virtual machine image. Rather than deploying a service to a full virtual machine, many developers deploy their services as Docker containers (or equivalent container technology) to the cloud. Virtual containers run inside a virtual machine; using a virtual container, you can segregate a single virtual machine into a series of self-contained processes that share 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 one or more instances of a service. 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 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.

Why not PaaS-based Microservices?

Earlier in the chapter we identified three types of cloud platforms (Infrastructure as a Service, Platform as a Service and Software as a Services). For this book, I've 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, I have chosen to remain vendor independent and deploy all parts of my application (including the servers).

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. Setting up and tuning the application server and the corresponding Java container are abstracted away from you. While this is 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 are 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 demonstrate 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).

1.9 Microservices are more than 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 than 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 don't have a microservice take on too much responsibility? Remember, properly sized, 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, multiple service instances can quickly start and shutdown?
- **Resilient.** How do we 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 takes a patterns-based approach as we answer the questions above. With a pattens-based approach, we layout common designs that can be used across different technology implementations. So while we have chosen to use Spring Boot and Spring Cloud to implement the patterns we are going to use in this book, there is nothing to keep you from taking the concepts presented here and use them with other technology platforms. Specifically, we 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.

1.9.1 Core microservice development pattern

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

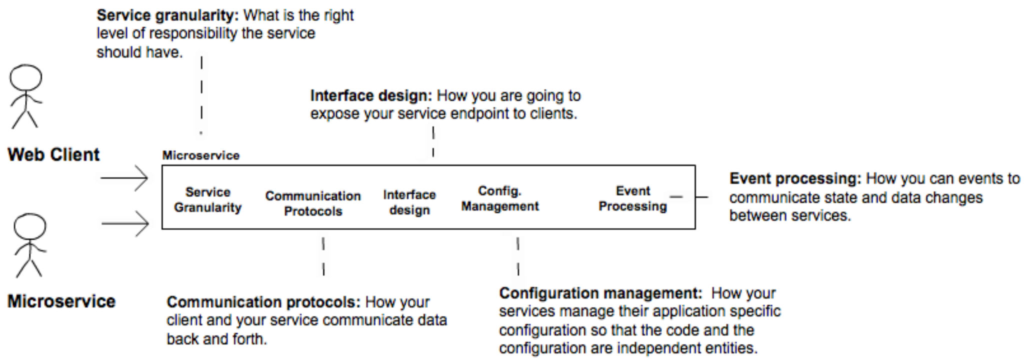

Service granularity: What is the right level of responsibility the service should have.
Interface design: How you are going to expose your service endpoint to clients.
Web Client
Microservice
Service Granularity
Communication Protocols
Interface design
Config. Management
Event Processing
Event processing: How you can events to communicate state and data changes between services.
Microservice
Communication protocols: How your client and your service communicate data back and forth.
Configuration management: How your services manage their application specific configuration so that the code and the configuration are independent entities.

Figure 1.5 – In building a simple service, we have to look beyond just writing the code and think about how the service will be consumed (the interface), how the service will be communicated with (the communication protocol) and how we will manage the configuration of the service as it is deployed to different environments

- **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 the 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 “dumb” data abstraction layer with no logic but that needed to access the data store. I cover service granularity in chapter 2.

- **Communication Protocols.** How will developers communicate with your service? We'll go into why JSON is the ideal choice for microservices and has become the most common choice for sending and receiving data to microservice. I cover communication protocols in chapter 2.
- **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. I cover interface design 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? I cover managing service configuration 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? I cover event processing between services in chapter 7.

1.9.2 Microservice routing patterns

The microservice routing patterns deal with how client application that wants to consume a microservice discovers the location of the service and is routed to it over. 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.

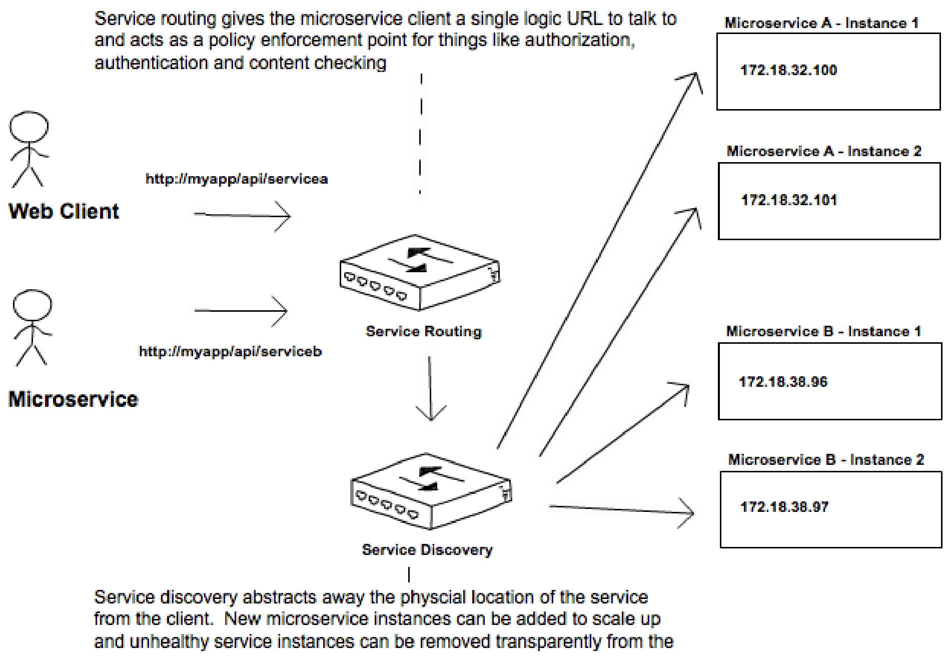

Service routing gives the microservice client a single logic URL to talk to and acts as a policy enforcement point for things like authorization, authentication and content checking
Web Client
http://myapp/api/servicea
Microservice
http://myapp/api/serviceb
Service Routing
Service Discovery
Service discovery abstracts away the physcial location of the service from the client. New microservice instances can be added to scale up and unhealthy service instances can be removed transparently from the
Microservice A - Instance 1
172.18.32.100
Microservice A - Instance 2
172.18.32.101
Microservice B - Instance 1
172.18.38.96
Microservice B - Instance 2
172.18.38.97

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

Service discovery and routing answer the question of "how do I get my client's 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? I cover service discovery 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 doesn't have to come up with their own solutions for providing routing to their services? I cover service routing in chapter 6.

In figure 1.3 service discovery and service routing 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. You can implement service routing without service discovery (even though its implementation is more difficult)

1.9.3 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:

- **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 slowly, it consumes 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 than 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?

Figure 1.7 shows how these patterns protect the consumer of service from being taken impacted when a service is misbehaving. I cover these four topics in chapter 5.

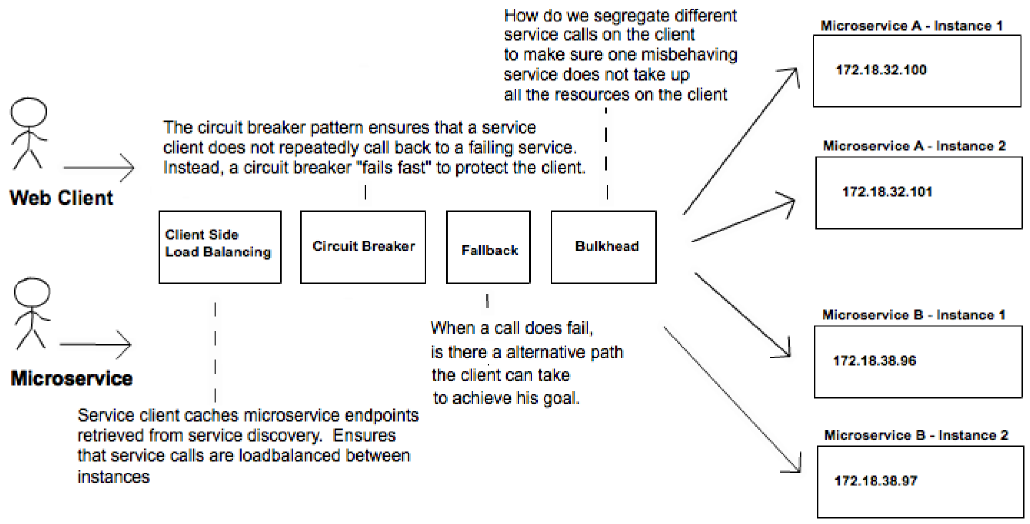

How do we segregate different service calls on the client to make sure one misbehaving service does not take up all the resources on the client
Microservice A - Instance 1
172.18.32.100
The circuit breaker pattern ensures that a service client does not repeatedly call back to a failing service. Instead, a circuit breaker "fails fast" to protect the client.
Microservice A - Instance 2
172.18.32.101
Web Client
Client Side Load Balancing
Circuit Breaker
Fallback
Bulkhead
Microservice B - Instance 1
172.18.38.96
When a call does fail, is there a alternative path the client can take to achieve his goal.
Microservice
Service client caches microservice endpoints retrieved from service discovery. Ensures that service calls are loadbalanced between instances
Microservice B - Instance 2
172.18.38.97

Figure 1.7 – With microservices, you must take care to protect the service caller from a poorly behaving services

1.9.4 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't allow "configuration drift" (something changes on a server after it has been deployed) to occur, as this can introduce instability 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 with good intentions. 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 occurs and everyone is left scratching their head wondering what's different between the lower environments in production. I've 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

An immutable infrastructure is a 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 or EAR to an already running piece of infrastructure. Instead we want to build and compile 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.

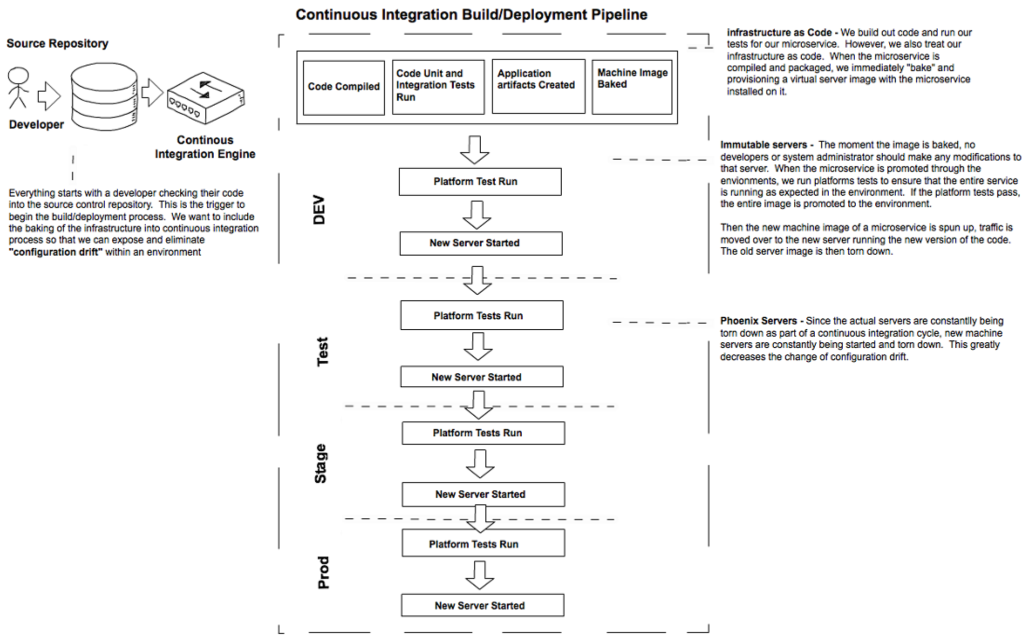

Continuous Integration Build/Deployment Pipeline
Source Repository
Developer
Continous Integration Engine
Everything starts with a developer checking their code into the source control repository. This is the trigger to begin the build/deployment process. We want to include the baking of the infrastructure into continuous integration process so that we can expose and eliminate **"configuration drift"** within an environment
Code Compiled
Code Unit and Integration Tests Run
Application artifacts Created
Machine Image Baked
DEV
Platform Test Run
New Server Started
Test
Platform Tests Run
New Server Started
Stage
Platform Tests Run
New Server Started
Prod
Platform Tests Run
New Server Started
Infrastructure as Code - We build out code and run our tests for our microservice. However, we also treat our infrastructure as code. When the microservice is compiled and packaged, we immediately "bake" and provisioning a virtual server image with the microservice installed on it.
Immutable servers - The moment the image is baked, no developers or system administrator should make any modifications to that server. When the microservice is promoted through the envionments, we run platforms tests to ensure that the entire service is running as expected in the environment. If the platform tests pass, the entire image is promoted to the environment.
Then the new machine image of a microservice is spun up, traffic is moved over to the new server running the new version of the code. The old server image is then torn down.
Phoenix Servers - Since the actual servers are constantly being torn down as part of a continuous integration cycle, new machine servers are constantly being started and torn down. This greatly decreases the change of configuration drift.

Figure 1.8 We want the 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 will 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 cover the following patterns and topics:

- **Build and Deployment Pipeline**. How do you create a repeatable build and deployment process that emphasizes 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 and stamp out configuration drift as quickly as possible before it is allowed to hit our environment upper environments like stage or production.

1.10 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 wraps 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 maps the patterns listed in the previous section to the Spring Cloud projects that implement them.

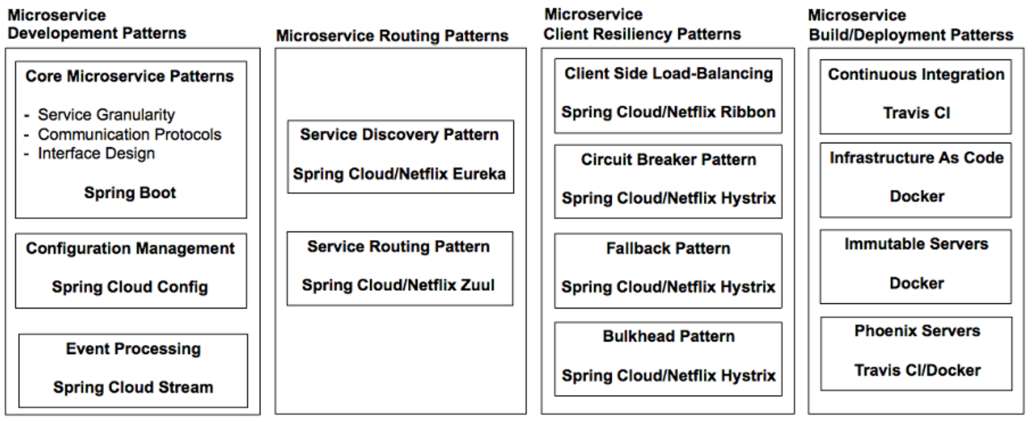

Microservice Developement Patterns
Core Microservice Patterns
- Service Granularity
- Communication Protocols
- Interface Design
Spring Boot
Configuration Management
Spring Cloud Config
Event Processing
Spring Cloud Stream
Microservice Routing Patterns
Service Discovery Pattern
Spring Cloud/Netflix Eureka
Service Routing Pattern
Spring Cloud/Netflix Zuul
Microservice Client Resiliency Patterns
Client Side Load-Balancing
Spring Cloud/Netflix Ribbon
Circuit Breaker Pattern
Spring Cloud/Netflix Hystrix
Fallback Pattern
Spring Cloud/Netflix Hystrix
Bulkhead Pattern
Spring Cloud/Netflix Hystrix
Microservice Build/Deployment Patterss
Continuous Integration
Travis CI
Infrastructure As Code
Docker
Immutable Servers
Docker
Phoenix Servers
Travis CI/Docker

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

Let's walkthrough these technologies in greater detail.

1.10.1 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 (GET, PUT, POST and DELETE) to URLs and the serialization of the JSON protocol to and from Java objects. It also simplifies the mapping of Java exceptions back to standard HTTP error codes.

1.10.2 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 its own property management repository, but also integrates with open source projects like:

- Git. Git (https://git-scm.com/) is an open source version control system that allows you to manage and track changes to any type of text file. Spring Cloud Config can integrate with a git-backed repository and read the applications configuration data out of the repository.
- Consul. Consul (https://www.consul.io/) is an open source service discovery tool that allows service instances to register themselves with the service. Service clients can then ask Consul where the service instances are located. Consul also includes key-value store based database that can be used by Spring Cloud Config to store application configuration data.
- Eureka. Eureka (https://github.com/Netflix/eureka) is an open source Netflix project that like Consul, offers similar service discovery capabilities. Eureka also has a key-value database that can be used with Spring Cloud Config.

1.10.3 Spring Cloud Service Discovery

With Spring Cloud service discovery, you can 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 than 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's service discovery engine.

1.10.4 Spring Cloud/Netflix Hystrix and Ribbon

Spring Cloud heavily integrates with Netflix open source projects. For microservice client resiliency patterns, Spring Cloud wraps the Netflix Hystrix libraries (https://github.com/Netflix/Hystrix)and Ribbon project (https://github.com/Netflix/Ribbon)and makes leveraging them from within your own microservices trivial to implement.

Using the Netflix Hystrix libraries, you can 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 makes it possible for a client to continue making service calls even if the service discovery agent is temporarily unavailable.

1.10.5 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 is a service gateway that that proxies service request and makes sure that all calls to your microservices go through a single "front door" before the targeted service is invoked. With this centralization of service calls with Zuul, you can enforce standard service policies like a security authorization authentication, content filtering, and routing rules.

1.10.6 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. Using Spring Cloud Stream, you can build intelligent microservices that can leverage asynchronous events as they occur in your application. With Spring Cloud Stream, you can quickly integrate your microservices with message brokers like RabbitMQ (https://www.rabbitmq.com/) and Kafka (http://kafka.apache.org/).

1.10.7 What about Provisioning?

For the provisioning implementations, we are going to make a technology shift. The Spring framework(s) are geared toward application development. The Spring frameworks (including Spring Cloud) don't have tools for creating a "build and deployment" pipeline. To implement a "build and deployment" pipeline 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.

1.11 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 couldn't fit all that goodness into a single code example at the beginning of the book.

In the code showing in listing 1.2, I 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     ❶
@EnableEurekaClient       ❷
public class Application {

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

    @HystrixCommand(threadPoolKey = "helloThreadPool")               ❸
public String helloRemoteServiceCall(String firstName,
                                                String lastName){
        ResponseEntity<String> restExchange =
               restTemplate.exchange(
                    "http://nameservice/name/{firstName}/{lastName}", ❹
                       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

❶ **Enables the service to use the Hystrix and Ribbon Libraries**

❷ **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**

❸ **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 walkthrough it. Keep in mind that listing 1.2 is an example 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 declare 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 is called, it will not be directly invoked. Instead, the method will be delegated to a thread pool managed by Hystrix. If the call takes too long (default is 1 second), Hystrix steps in and interrupts 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 `helloRemoteServiceCall` method. The presence of the `@EnableEurekaClient` has told Spring Boot that we are going to use 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 of the "name" service instances. As a consumer of the service, our code never has to know where that service is located.

Also, the `RestTemplate` class is leveraging Netflix's Ribbon library. Ribbon will retrieve a list of all of the physical endpoints associated with a service. Every time the service is called by the client, it round robins 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 number 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 used outside of Spring Cloud, can be complex and obtuse to setup. Spring Cloud simplifies their use to literally nothing more than a few simple Spring Cloud annotations and configuration entries.

1.12 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 have structured 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. 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, "onpremise" 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 need some reorganization as the team re-evaluates their responsibilities 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.

1.13 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 is 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 make it possible for you to build microservices quickly with nothing more than 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.