John Carnell

MANNING

www.allitebooks.com

*Spring Microservices in Action*

**Licensed to <null>** www.allitebooks.com

**Licensed to <null>** www.allitebooks.com

*Spring Microservices in Action*

JOHN CARNELL

MANNING SHELTER ISLAND

**Licensed to <null>** www.allitebooks.com

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department Manning Publications Co. 20 Baldwin Road PO Box 761 Shelter Island, NY 11964 Email: orders@manning.com

©2017 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning’s policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co. Acquisition editor: Greg Wild 20 Baldwin Road Development editor: Marina Michaels PO Box 761 Technical development editor: Raphael Villela Shelter Island, NY 11964 Copyeditor: Katie Petito

Proofreader: Melody Dolab Technical proofreader: Joshua White

Review editor: Aleksandar Dragosavljevic

Typesetter: Marija Tudor Cover designer: Marija Tudor

ISBN 9781617293986 Printed in the United States of America 1 2 3 4 5 6 7 8 9 10 – EBM – 22 21 20 19 18 17

**Licensed to <null>** www.allitebooks.com

*To my brother Jason, who even in his darkest moments showed me the true meaning of strength and dignity. You are a role model as a brother, husband, and father.*

**Licensed to <null>**

**vi**

**Licensed to <null>**

*brief contents*

*1* ■ Welcome to the cloud, Spring 1 *2* ■ Building microservices with Spring Boot 35 *3* ■ Controlling your configuration with Spring Cloud

configuration server 64 *4* ■ On service discovery 96 *5* ■ When bad things happen: client resiliency patterns with

Spring Cloud and Netflix Hystrix 119 *6* ■ Service routing with Spring Cloud and Zuul 153 *7* ■ Securing your microservices 192 *8* ■ Event-driven architecture with Spring Cloud Stream 228 *9* ■ Distributed tracing with Spring Cloud Sleuth and Zipkin 259 *10* ■ Deploying your microservices 288

**vii**

**Licensed to <null>**

BRIEF CONTENTS **viii**

**Licensed to <null>**

*contents*

*preface xv acknowledgments xvii about this book xix about the author xxii about the cover illustration xxiii 1* ***Welcome*** 1.1 ***to the cloud, Spring 1***

What’s a microservice? 2 1.2 What is Spring and why is it relevant to microservices? 5 1.3 What you’ll learn in this book 6 1.4 Why is this book relevant to you? 7 1.5 Building a microservice with Spring Boot 8 1.6 Why change the way we build applications? 12 1.7 What exactly is the cloud? 13 1.8 Why the cloud and microservices? 15 1.9 Microservices are more than writing the code 17

*Core microservice development pattern 19* ■ *Microservice routing patterns 20* ■ *Microservice client resiliency patterns 21 Microservice security patterns 23* ■ *Microservice logging and tracing patterns 24* ■ *Microservice build/deployment patterns 25* 1.10 Using Spring Cloud in building your microservices 26

*Spring Boot 28* ■ *Spring Cloud Config 28* ■ *Spring Cloud service discovery 28* ■ *Spring Cloud/Netflix Hystrix and*

**ix**

**Licensed to <null>**

CONTENTS **x**

*Ribbon 29* ■ *Spring Cloud/Netflix Zuul 29* ■ *Spring Cloud Stream 29* ■ *Spring Cloud Sleuth 29* ■ *Spring Cloud Security 30* ■ *What about provisioning? 30* 1.11 Spring Cloud by example 30 1.12 Making sure our examples are relevant 33 1.13 Summary 33 *2* ***Building microservices with Spring Boot 35***

2.1 The architect’s story: designing the microservice

architecture 38 *Decomposing the business problem 38* ■ *Establishing service granularity 41* ■ *Talking to one another: service interfaces 43* 2.2 When not to use microservices 44

*Complexity of building distributed systems 44* ■ *Server sprawl 44* ■ *Type of application 44* ■ *Data transformations and consistency 45* 2.3 The developer’s tale: building a microservice with Spring

Boot and Java 45 *Getting started with the skeleton project 46* ■ *Booting your Spring Boot application: writing the Bootstrap class 47* ■ *Building the doorway into the microservice: the Spring Boot controller 48* 2.4 The DevOps story: building for the rigors of runtime 53

*Service assembly: packaging and deploying your microservices 56*

*Service bootstrapping: managing configuration of your microservices 58* ■ *Service registration and discovery: how clients communicate with your microservices 59* ■ *Communicating a microservice’s health 60* 2.5 Pulling the perspectives together 62 2.6 Summary 63 *3* ***Controlling your configuration with Spring Cloud***

***configuration server 64*** 3.1 On managing configuration (and complexity) 65

*Your configuration management architecture 67* ■ *Implementation choices 69* 3.2 Building our Spring Cloud configuration server 70

*Setting up the Spring Cloud Config Bootstrap class 74 Using Spring Cloud configuration server with the filesystem 75*

**Licensed to <null>**

CONTENTS **xi**

3.3 Integrating Spring Cloud Config with a Spring Boot client 77

*Setting up the licensing service Spring Cloud Config server dependencies 79* ■ *Configuring the licensing service to use Spring Cloud Config 79* ■ *Wiring in a data source using Spring Cloud configuration server 83* ■ *Directly Reading Properties using the @Value Annotation 86* ■ *Using Spring Cloud configuration server with Git 87* ■ *Refreshing your properties using Spring Cloud configuration server 88* 3.4 Protecting sensitive configuration information 89

*Download and install Oracle JCE jars needed for encryption 90 Setting up an encryption key 91* ■ *Encrypting and decrypting a property 91* ■ *Configure microservices to use encryption on the client side 93* 3.5 Closing thoughts 95 3.6 Summary 95 *4* ***On service discovery 96***

4.1 Where’s my service? 97 4.2 On service discovery in the cloud 100

*The architecture of service discovery 100* ■ *Service discovery in action using Spring and Netflix Eureka 103* 4.3 Building your Spring Eureka Service 105 4.4 Registering services with Spring Eureka 107 4.5 Using service discovery to look up a service 111

*Looking up service instances with Spring DiscoveryClient 112 Invoking services with Ribbon-aware Spring RestTemplate 114 Invoking services with Netflix Feign client 116* 4.6 Summary 118 *5* ***When bad things happen: client resiliency patterns with***

***Spring Cloud and Netflix Hystrix 119*** 5.1 What are client-side resiliency patterns? 120

*Client-side load balancing 121* ■ *Circuit breaker 122 Fallback processing 122* ■ *Bulkheads 122* 5.2 Why client resiliency matters 123 5.3 Enter Hystrix 126 5.4 Setting up the licensing server to use Spring Cloud and

Hystrix 127

**Licensed to <null>**

CONTENTS **xii**

5.5 Implementing a circuit breaker using Hystrix 128

*Timing out a call to the organization microservice 131 Customizing the timeout on a circuit breaker 132* 5.6 Fallback processing 133 5.7 Implementing the bulkhead pattern 136 5.8 Getting beyond the basics; fine-tuning Hystrix 138

*Hystrix configuration revisited 142* 5.9 Thread context and Hystrix 144

*ThreadLocal and Hystrix 144* ■ *The HystrixConcurrencyStrategy in action 147* 5.10 Summary 151 *6* ***Service routing with Spring Cloud and Zuul 153***

6.1 What is a services gateway? 154 6.2 Introducing Spring Cloud and Netflix Zuul 157

*Setting up the Zuul Spring Boot project 157* ■ *Using Spring Cloud annotation for the Zuul service 157* ■ *Configuring Zuul to communicate with Eureka 158* 6.3 Configuring routes in Zuul 159

*Automated mapping routes via service discovery 159 Mapping routes manually using service discovery 161 Manual mapping of routes using static URLs 165 Dynamically reload route configuration 168* ■ *Zuul and service timeouts 169* 6.4 The real power of Zuul: filters 169 6.5 Building your first Zuul pre-filter generating correlation

IDs 173 *Using the correlation ID in your service calls 176* 6.6 Building a post filter receiving correlation IDs 182 6.7 Building a dynamic route filter 184

*Building the skeleton of the routing filter 186* ■ *Implementing the run() method 187* ■ *Forwarding the route 188* ■ *Pulling it all together 190* 6.8 Summary 191 *7* ***Securing your microservices 192***

7.1 Introduction to OAuth2 193

**Licensed to <null>**

CONTENTS **xiii**

7.2 Starting small: using Spring and OAuth2 to protect a

single endpoint 195 *Setting up the EagleEye OAuth2 authentication service 196 Registering client applications with the OAuth2 service 197 Configuring EagleEye users 200* ■ *Authenticating the user 202* 7.3 Protecting the organization service using OAuth2 205

*Adding the Spring Security and OAuth2 jars to the individual services 205* ■ *Configuring the service to point to your OAuth2 authentication service 206* ■ *Defining who and what can access the service 207* ■ *Propagating the OAuth2 access token 210* 7.4 JavaScript Web Tokens and OAuth2 213

*Modifying the authentication service to issue JavaScript Web Tokens 214* ■ *Consuming JavaScript Web Tokens in your microservices 218* ■ *Extending the JWT Token 220 Parsing a custom field out of a JavaScript token 222* 7.5 Some closing thoughts on microservice security 224 7.6 Summary 227 *8* ***Event-driven architecture with Spring Cloud Stream 228***

8.1 The case for messaging, EDA, and microservices 229

*Using synchronous request-response approach to communicate state change 230* ■ *Using messaging to communicate state changes between services 233* ■ *Downsides of a messaging architecture 235* 8.2 Introducing Spring Cloud Stream 236

*The Spring Cloud Stream architecture 237* 8.3 Writing a simple message producer and consumer 238

*Writing the message producer in the organization service 239 Writing the message consumer in the licensing service 244 Seeing the message service in action 247* 8.4 A Spring Cloud Stream use case: distributed caching 249

*Using Redis to cache lookups 250* ■ *Defining custom channels 256 Bringing it all together: clearing the cache when a message is received 257* 8.5 Summary 258 *9* ***Distributed tracing with Spring Cloud Sleuth and Zipkin 259***

9.1 Spring Cloud Sleuth and the correlation ID 260

*Adding Spring Cloud sleuth to licensing and organization 261 Anatomy of a Spring Cloud Sleuth trace 262*

**Licensed to <null>**

CONTENTS **xiv**

9.2 Log aggregation and Spring Cloud Sleuth 263

*A Spring Cloud Sleuth/Papertrail implementation in action 265 Create a Papertrail account and configure a syslog connector 267 Redirecting Docker output to Papertrail 268* ■ *Searching for Spring Cloud Sleuth trace IDs in Papertrail 270* ■ *Adding the correlation ID to the HTTP response with Zuul 272* 9.3 Distributed tracing with Open Zipkin 274

*Setting up the Spring Cloud Sleuth and Zipkin dependencies 275 Configuring the services to point to Zipkin 275* ■ *Installing and configuring a Zipkin server 276* ■ *Setting tracing levels 278 Using Zipkin to trace transactions 278* ■ *Visualizing a more complex transaction 281* ■ *Capturing messaging traces 282 Adding custom spans 284* 9.4 Summary 287 *10* ***Deploying your microservices 288***

10.1 EagleEye: setting up your core infrastructure in the cloud 290

*Creating the PostgreSQL database using Amazon RDS 293 Creating the Redis cluster in Amazon 296* ■ *Creating an ECS cluster 298* 10.2 Beyond the infrastructure: deploying EagleEye 302

*Deploying the EagleEye services to ECS manually 303* 10.3 The architecture of a build/deployment pipeline 305 10.4 Your build and deployment pipeline in action 309 10.5 Beginning your build deploy/pipeline: GitHub and Travis

CI 311 10.6 Enabling your service to build in Travis CI 312

*Core build run-time configuration 315* ■ *Pre-build tool installations 318* ■ *Executing the build 320* ■ *Tagging the source control code 320* ■ *Building the microservices and creating the Docker images 321* ■ *Pushing the images to Docker Hub 322 Starting the services in Amazon ECS 323* ■ *Kicking off the platform tests 323* 10.7 Closing thoughts on the build/deployment pipeline 325 10.8 Summary 325

*appendix A Running a cloud on your desktop 327 appendix B OAuth2 grant types 336*

*index 345*

**Licensed to <null>**

*preface*

It’s ironic that in writing a book, the last part of the book you write is often the begin- ning of the book. It’s also often the most difficult part to put down on paper. Why? Because you have to explain to everyone why you’re so passionate about a subject that you spent the last one and a half years of your life writing a book about it. It’s hard to articulate why anyone would spend such a large amount of time on a technical book. One rarely writes software books for the money or the fame.

Here’s the reason why I wrote this book: I love writing code. It’s a calling for me and it’s also a creative activity—akin to drawing, painting, or playing an instrument. Those outside the field of software development have a hard time understanding this. I especially like building distributed applications. For me, it’s an amazing thing to see an application work across dozens (even hundreds) of servers. It’s like watching an orchestra playing a piece of music. While the final product of an orchestra is beauti- ful, the making of it is often a lot of hard work and requires a significant amount of practice. The same goes for writing a massively distributed application.

Since I entered the software development field 25 years ago, I’ve watched the industry struggle with the “right” way to build distributed applications. I’ve seen dis- tributed service standards such as CORBA rise and fall. Monstrously big companies have tried to push big and, often, proprietary protocols. Anyone remember Micro- soft’s Distributed Component Object Model (DCOM) or Oracle’s J2EE’s Enterprise Java Beans 2 (EJB)? I watched as technology companies and their followers rushed to build service-oriented architectures (SOA) using heavy XML-based schemas.

In each case, these approaches for building distributed systems often collapsed under their own weight. I’m not saying that these technologies weren’t used to build some very powerful applications. The reality is that they couldn’t keep up with the

**xv**

**Licensed to <null>**

PREFACE **xvi**

demand of the users. Ten years ago, smartphones were just being introduced to the market and cloud computing was in the earliest stage of infancy. Also, the standards and technology for distributed application development were too complicated for the average developer to understand and easily use in practice. Nothing speaks truth in the software development industry like written code. When the standards get in the way of this, the standards quickly get discarded.

When I first heard of the microservices approach to building applications I was more than a little skeptical. “Great, another silver-bullet approach to building distrib- uted applications,” I thought. However, as I started diving into the concepts, I realized the simplicity of microservices could be a game changer. A microservice architecture focuses on building small services that use simple protocols (HTTP and JSON) to com- municate. That’s it. You can write a microservice with nearly any programming lan- guage. There’s beauty in this simplicity.

However, while building an individual microservice is easy, operationalizing and scaling it is difficult. Getting hundreds of small distributed components to work together and then building a resilient application from them can be incredibly diffi- cult to do. In distributed computing, failure is a fact of life and how your application deals with it is incredibly difficult to get right. To paraphrase my colleagues Chris Miller and Shawn Hagwood: “If it’s not breaking once in a while, you’re not building.” It’s these failures that inspired me to write this book. I hate to build things from scratch when I don’t have to. The reality is that Java is the lingua franca for most appli- cation development efforts, especially in the enterprise. The Spring framework has for many organizations become the de facto framework for most application develop- ment. I’d already been doing application development in Java for almost 20 years (I remember the Dancing Duke applet) and Spring for almost 10 years. As I began my microservices journey, I was delighted and excited to watch the emergence of Spring Cloud.

The Spring Cloud framework provides out-of-the-box solutions for many of the common development and operational problems you’ll run into as a microservice developer. Spring Cloud lets you use only the pieces you need and minimizes the amount of work you need to do to build and deploy production-ready Java micro- services. It does this by using other battle-hardened technologies from companies and groups such as Netflix, HashiCorp, and the Apache foundation.

I’ve always considered myself an average developer who, at the end of the day, has deadlines to meet. That’s why I undertook the project of writing this book. I wanted a book that I could use in my day-to-day work. I wanted something with direct (and hopefully) straightforward code examples. I always want to make sure that the mate- rial in this book can be consumed as individual chapters or in its entirety. I hope you find this book useful and I hope you enjoy reading it as much as I enjoyed writing it.

**Licensed to <null>**

**xvii***acknowledgments* As I sit down to write these acknowledgments, I can’t help but think back to 2014 when I ran my first marathon. Writing a book is a lot like running a marathon. Writing the proposal and the outline for the book is much like the training process. It gets your thoughts in shape, it focuses you for what’s ahead and, yes, near the end of the process, it can be more than a little tedious and brutal.

When you start writing the book, it’s a lot like race day. You start the marathon excited and full of energy. You know you’re trying to do something bigger than any- thing you might have done before and it’s both exciting and nerve-wracking. This is what you’ve trained for, but at the same time, there’s always that small voice of doubt in the back of your mind that says you won’t finish what you started.

What I’ve learned from running is that races aren’t completed one mile at a time. Instead, they’re run one foot in front of the other. The miles run are the sum of the individual footsteps. When my children are struggling with something, I laugh and ask them, “How do you write a book? One word, one single step at a time.” They usually roll their eyes, but in the end there’s no other way around this indisputable and iron- clad law. However, when you run a marathon, you might be the one running the race, but you’re never running it alone. There’s a whole team of people there to give you sup- port, time, and advice along the way. It has been the same experience writing this book. I’d like to start by thanking Manning for the support they gave me in writing this book. It started with Greg Wild, my acquisitions editor, who patiently worked with me as I refined the core concepts in this book and guided me through the proposal pro- cess. Along the way, Marina Michaels, my development editor, kept me honest and

**Licensed to <null>**

ACKNOWLEDGMENTS **xviii**

challenged me to become a better author. I’d also like to thank Raphael Villela and Joshua White, my technical editors, who constantly checked my work and ensured the overall quality of the examples and the code I produced. I’m extremely grateful for the time, talent, and commitment each of these individuals put into into the overall project. I’d also like to thank the reviewers who provided feedback on the manuscript throughout the writing and development process: Aditya Kumar, Adrian M. Rossi, Ashwin Raj, Christian Bach, Edgar Knapp, Jared Duncan, Jiri Pik, John Guthrie, Mirko Bernardoni, Paul Balogh, Pierluigi Riti, Raju Myadam, Rambabu Posa, Sergey Evsikov, and Vipul Gupta.

I want to close these acknowledgments with a deep sense of thanks for the love and time my family has given me in working on this project. To my wife Janet, you have been my best friend and the love of my life. When I’m tired and want to give up, I only have to listen for the sound of your footsteps next to me to know that you’re always running beside me, never telling me no, and always pushing me forward.

To my son Christopher, you’re growing up to be an incredible young man. I can- not wait for the day when you truly discover your passion, because there will be noth- ing in this world that can stop you from reaching your goals.

To my daughter Agatha, I’d give all the money I have to see the world through your eyes for just 10 minutes. The experience would make me a better author and more importantly a better person. Your intellect, your power of observation, and cre- ativity humble me.

To my four-year-old son, Jack: Buddy, thank you being patient with me whenever I said, “I can’t play right now because Daddy has to work on the book.” You always make me laugh and you make this whole family complete. Nothing makes me happier than when I see you being the jokester and playing with everyone in the family.

My race with this book is done. Like my marathon, I’ve left nothing on the table in writing this book. I have nothing but gratitude for the Manning team and the MEAP readers who bought this book early and gave me so much valuable feedback. I hope in the end that you enjoy this book as much as I enjoyed writing it. Thank you.

**Licensed to <null>**

*about this book*

*Spring Microservices in Action* was written for the practicing Java/Spring developer who needs hands-on advice and examples of how to build and operationalize microservice- based applications. When I wrote this book, I wanted it to be based around core microservice patterns that aligned with Spring Boot and Spring Cloud examples that demonstrated the patterns in action. As such, you’ll find specific microservice design patterns discussed in almost every chapter, along with examples of the patterns imple- mented using Spring Boot and Spring Cloud.

***You should read this book if***

▪ You’re a Java developer who has experience building distributed applications (1-3 years).

▪ You have a background in Spring (1+ years).

▪ You’re interested in learning how to build microservice-based applications.

▪ You’re interested in how you can use microservices for building cloud-based applications.

▪ You want to know if Java and Spring are relevant technologies for building microservice-based applications.

▪ You’re interested in seeing what goes into deploying a microservice-based appli- cation to the cloud.

***How this book is organized*** *Spring Microservices in Action* consists of 10 chapters and two appendixes:

▪ Chapter 1 introduces you to why the microservices architecture is an important and relevant approach to building applications, especially cloud-based applications.

**xix**

**Licensed to <null>**

ABOUT THIS BOOK **xx**

▪ Chapter 2 walks you through how to build your first REST-based microservice using Spring Boot. This chapter will guide you in how to look at your microser- vices through the eyes of an architect, an application engineer, and a DevOps engineer.

▪ Chapter 3 introduces you to how to manage the configuration of your microser- vices using Spring Cloud Config. Spring Cloud Config helps you guarantee that your service’s configuration information is centralized in a single repository, versioned and repeatable across all instances of your services.

▪ Chapter 4 introduces you to one of the first microservice routing patterns: ser- vice discovery. In this chapter, you’ll learn how to use Spring Cloud and Net- flix’s Eureka service to abstract away the location of your services from the clients consuming them.

▪ Chapter 5 is all about protecting the consumers of your microservices when one or more microservice instances is down or in a degraded state. This chapter will demonstrate how to use Spring Cloud and Netflix Hystrix (and Netflix Ribbon) to implement client-side load balancing of calls, the circuit breaker pattern, the fallback pattern, and the bulkhead pattern.

▪ Chapter 6 covers the microservice routing pattern: the service gateway. Using Spring Cloud with Netflix’s Zuul server, you’ll build a single entry point for all microservices to be called through. We’ll discuss how to use Zuul’s filter API to build policies that can be enforced against all services flowing through the ser- vice gateway.

▪ Chapter 7 covers how to implement service authentication and authorization using Spring Cloud security and OAuth2. We’ll cover the basics of setting up an OAuth2 service to protect your services and also how to use JavaScript Web Tokens (JWT) in your OAuth2 implementation.

▪ Chapter 8 looks at how you can introduce asynchronous messaging into your microservices using Spring Cloud Stream and Apache Kafka.

▪ Chapter 9 shows how to implement common logging patterns such as log corre- lation, log aggregation, and tracing using Spring Cloud Sleuth and Open Zipkin.

▪ Chapter 10 is the cornerstone project for the book. You’ll take the services you’ve built in the book and deploy them to Amazon Elastic Container Service (ECS). We’ll also discuss how to automate the build and deployment of your microservices using tools such as Travis CI.

▪ Appendix A covers how to set up your desktop development environment so that you can run all the code examples in this book. This appendix covers how the local build process works and also how to start up Docker locally if you want to run the code examples locally.

▪ Appendix B is supplemental material on OAuth2. OAuth2 is an extremely flexi- ble authentication model, and this chapter provides a brief overview of the dif- ferent manners in which OAuth2 can be used to protect an application and its corresponding microservices.

**Licensed to <null>**

ABOUT THIS BOOK **xxi**

***About the code***

*Spring Microservices in Action* includes code in every chapter. All code examples are avail- able in my GitHub repository, and each chapter has its own repository. You can find an overview page with links to each chapter’s code repository at https://github.com/ carnellj/spmia\_overview. A zip containing all source code is also available from the publisher’s website at www.manning.com/books/spring-microservices-in-action.

All code in this book is built to run on Java 8 using Maven as the main build tool. Please refer to appendix A of this book for full details on the software tools you’ll need to compile and run the code examples.

One of the core concepts I followed as I wrote this book was that the code exam- ples in each chapter should run independently of those in the other chapters. As such, every service we create for a chapter builds to a corresponding Docker image. When code from previous chapters is used, it’s included as both source and a built Docker image. We use Docker compose and the built Docker images to guarantee that you have a reproducible run-time environment for every chapter.

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text. Sometimes code is also **in bold** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we’ve added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this wasn’t enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

***Author Online***

Purchase of *Spring Microservices in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/books /spring-microservices-in-action. This page provides information on how to get on the forum once you’re registered, what kind of help is available, and the rules of conduct on the forum.

Manning’s commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contributions to the AO remain voluntary (and unpaid). We suggest you ask the author challenging questions, lest his interest stray!

**Licensed to <null>**

**xxii***about the author* JOHN CARNELL is a senior cloud engineer at Genesys, where he works in Genesys’s PureCloud division. John spends the major- ity of his day hands-on building telephony-based microservices using the AWS platform. His day-to-day job centers on designing and building microservices across a number of technology plat- forms including Java, Clojure, and Go.

John is a prolific speaker and writer. He regularly speaks at local user groups and has been a regular speaker on “The No Fluff Just Stuff Software Symposium.” Over the last 20 years, John has authored, co- authored, and been a technical reviewer for a number of Java-based technology books and industry publications.

John holds a Bachelor of the Arts (BA) from Marquette University and a Masters of Business Administration (MBA) from the University of Wisconsin Oshkosh.

John is a passionate technologist and is constantly exploring new technologies and programming languages. When John isn’t speaking, writing, or coding, he lives with his wife Janet, his three children, Christopher, Agatha, and Jack, and yes, his dog Vader, in Cary, North Carolina.

During his free time (which there’s very little of) John runs, chases after his chil- dren, and studies Filipino martial arts.

John can be reached at john\_carnell@yahoo.com.

**Licensed to <null>**

*about the cover illustration*

The figure on the cover of *Spring Microservices in Action* is captioned a “A Man from Croatia.” This illustration is taken from a recent reprint of Balthasar Hacquet’s *Images and Descriptions of Southwestern and Eastern Wenda, Illyrians, and Slavs*, published by the Ethnographic Museum in Split, Croatia, in 2008. Hacquet (1739–1815) was an Aus- trian physician and scientist who spent many years studying the botany, geology, and ethnography of many parts of the Austrian Empire, as well as the Veneto, the Julian Alps, and the western Balkans, inhabited in the past by peoples of the Illyrian tribes. Hand drawn illustrations accompany the many scientific papers and books that Hac- quet published.

The rich diversity of the drawings in Hacquet's publications speaks vividly of the uniqueness and individuality of the eastern Alpine and northwestern Balkan regions just 200 years ago. This was a time when the dress codes of two villages separated by a few miles identified people uniquely as belonging to one or the other, and when members of a social class or trade could be easily distinguished by what they were wearing. Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another, and today the inhabitants of the picturesque towns and villages in the Slovenian Alps or Balkan coastal towns are not readily distinguishable from the resi- dents of other parts of Europe.

We at Manning celebrate the inventiveness, the initiative, and the fun of the com- puter business with book covers based on costumes from two centuries ago, brought back to life by illustrations such as this one.

**xxiii**

**Licensed to <null>**

ABOUT THE COVER ILLUSTRATION **xxiv**

**Licensed to <null>**

*Welcome to the cloud, Spring*

***This chapter covers***

▪ Understanding microservices and why companies use them

▪ Using Spring, Spring Boot, and Spring Cloud for building microservices

▪ Learning why the cloud and microservices are relevant to microservice-based applications

▪ Building microservices involves more than building service code

▪ Understanding the parts of cloud-based development

▪ Using Spring Boot and Spring Cloud in microservice development

The one constant in the field of software development is that we as software devel- opers 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, causing us to reevalu- ate how we build and deliver solutions for our customers. One example of this churn is the rapid adoption by many organizations of building applications using

**1**

**Licensed to <null>**

**2** CHAPTER 1 ***Welcome to the cloud, Spring***

microservices. Microservices are distributed, loosely coupled software services that carry out a small number of well-defined tasks.

This book introduces you to the microservice architecture and why you should consider building your applications with them. We’re going to look at how to build microservices using Java and two Spring framework projects: Spring Boot and Spring Cloud. If you’re a Java developer, Spring Boot and Spring Cloud will provide an easy migration path from building traditional, monolithic Spring applications to microser- vice applications that can be deployed to the cloud.

***1.1 What’s a microservice?***

Before the concept of microservices evolved, most web-based applications were built using a monolithic architectural style. In a monolithic architecture, an application is delivered as a single deployable software artifact. All the UI (user interface), business, and database access logic are packaged together into a single application artifact and deployed to an application server.

While an application might be a deployed as a single unit of work, most of the time there will be multiple development teams working on the application. Each develop- ment team will have their own discrete pieces of the application they’re responsible for and oftentimes specific customers they’re serving with their functional piece. For example, when I worked at a large financial services company, we had an in-house, custom-built customer relations management (CRM) application that involved the coordination of multiple teams including the UI, the customer master, the data ware- house, and the mutual funds team. Figure 1.1 illustrates the basic architecture of this application.

The problem here is that as the size and complexity of the monolithic CRM appli- cation grew, the communication and coordination costs of the individual teams work- ing on the application didn’t scale. Every time an individual team needed to make a change, the entire application had to be rebuilt, retested and redeployed.

The concept of a microservice originally crept into the software development com- munity’s consciousness around 2014 and was a direct response to many of the chal- lenges of trying to scale both technically and organizationally large, monolithic applications. Remember, a microservice is a small, loosely coupled, distributed service. Microservices allow you to take a large application and decompose it into easy-to- manage components with narrowly defined responsibilities. Microservices help combat the traditional problems of complexity in a large code base by decomposing the large code base down into small, well-defined pieces. The key concept you need to embrace as you think about microservices is decomposing and unbundling the functionality of

**Licensed to <null>**

**3 *What’s a microservice?***

**Each team has their own areas of responsibity with their own requirements and delivery demands. All their work is synchronized into a single code base.**

Java application server (JBoss, Websphere, WebLogic, Tomcat)

WAR

Mutual funds team

Spring services MVC

Continuous integration pipeline

Typical Spring-based web applications

Single source code Customer master

repository

team

Spring data

Data warehousing team

Mutual funds

Customer master database

database

UI team

**The entire application also has knowledge of and access to all of the data sources used within the application.**

Figure 1.1 Monolithic applications force multiple development teams to artificially synchronize their delivery because their code needs to be built, tested, and deployed as an entire unit.

**Licensed to <null>**

Data warehouse

**4** CHAPTER 1 ***Welcome to the cloud, Spring***

your applications so they’re completely independent of one another. If we take the CRM application we saw in figure 1.1 and decompose it into microservices, it might look like what’s shown in figure 1.2.

Looking at figure 1.2, you can see that each functional team completely owns their service code and service infrastructure. They can build, deploy, and test indepen- dently of each other because their code, source control repository, and the infrastruc- ture (app server and database) are now completely independent of the other parts of the application.

Continuous integration pipeline

Customer master source code repository

Mutual funds microservice

Mutual funds Mutual funds team source code repository Customer master

team

Data warehousing team

UI team

Mutual funds database

Continuous

Customer integration

master pipeline

microservice

Data warehouse source code repository

Customer master database

Continuous

Data integration

warehouse pipeline

microservice

UI source code repository

Data warehouse

Continuous integration pipeline

Figure 1.2 Using a microservice architecture our CRM application would be decomposed into a set of microservices completely independent of each other, allowing each development team to move at their own pace.

**Licensed to <null>**

UI web application

**Invokes all business logic as REST-based service calls**

**5 *What is Spring and why is it relevant to microservices?***

A microservice architecture has the following characteristics:

▪ Application logic is broken down into small-grained components with well- defined boundaries of responsibility that coordinate to deliver a solution.

▪ Each component has a small domain of responsibility and is deployed com- pletely 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 prin- ciples, not standards) and employ lightweight communication protocols such as HTTP and JSON (JavaScript Object Notation) for exchanging data between the service consumer and service provider.

▪ The underlying technical implementation of the service is irrelevant because the applications always communicate with a technology-neutral protocol (JSON is the most 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 toward a single goal such as delivering an application, but each team is responsible only for the services on which they’re 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 deployable to the cloud. Once the services are in the cloud, their small size makes it easy to start up large numbers of instances of the same service, and suddenly your applications become more scalable and, with forethought, more resilient.

***1.2 What is Spring and why is it relevant to microservices?***

Spring has become the de facto development framework for building Java-based appli- cations. At its core, Spring is based on the concept of dependency injection. In a nor- mal Java application, the application is decomposed into classes where each class often has explicit linkages to other classes in the application. The linkages are the invocation of a class constructor directly in the code. Once the code is compiled, these linkage points can’t be changed.

This is problematic in a large project because these external linkages are brittle and making a change can result in multiple downstream impacts to other code. A depen- dency injection framework, such as Spring, allows you to more easily manage large Java projects by externalizing the relationship between objects within your application through convention (and annotations) rather than those objects having hard-coded knowledge about each other. Spring sits as an intermediary between the different Java

**Licensed to <null>**

**6** CHAPTER 1 ***Welcome to the cloud, Spring***

classes of your application and manages their dependencies. Spring essentially lets you assemble your code together like a set of Lego bricks that snap together.

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 a way to building applications using the J2EE stack. The J2EE stack, while powerful, was considered by many to be bloatware, with many features that were never used by application development teams. Further, a J2EE application forced you to use a full- blown (and heavy) Java application server to deploy your applications.

What’s amazing about the Spring framework and a testament to its development community is its ability to stay relevant and reinvent 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 models where services were being built as small, distributed services that could be easily deployed to the cloud. In response to this shift, the Spring devel- opment 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 the “enterprise” features found in Spring and instead delivers a framework geared toward Java-based, REST-oriented (Representational State Transfer)1 microservices. With a few simple annotations, a Java developer can quickly build a REST microservice that can be packaged 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 should embrace the use of the HTTP verbs (GET, POST, PUT, and DELETE) to represent the core actions of the service and use a lightweight web-oriented data serialization protocol, such as JSON, for requesting and receiving data from the service.

Because 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 it simple to operationalize and deploy microservices to a private or public cloud. Spring Cloud wraps several popular cloud-management microservice frameworks under a common framework and makes the use and deployment of these technologies as easy to use as annotating your code. I cover the different components within Spring Cloud later in this chapter.

***1.3 What you’ll 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

1 While we cover REST later in chapter 2, it’s 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.

**Licensed to <null>**

**7 *Why is this book relevant to you?***

cloud such as 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 framework

▪ The core operational patterns that need to be in place to support microservice applications, particularly a cloud-based application

▪ How you can use Spring Cloud to implement these operational patterns

▪ How to take what you’ve learned and build a deployment pipeline that can be used to deploy your services to a private, internally managed cloud or a public cloud provider

By the time you’re done reading this book, you should have the knowledge needed to build and deploy a Spring Boot-based microservice. You’ll also understand the key design decisions need to operationalize your microservices. You’ll understand how service configuration management, service discovery, messaging, logging and tracing, and security all fit together to deliver a robust microservices environment. Finally, you’ll see how your microservices can be deployed within a private or public cloud.

***1.4 Why is this book relevant to you?***

If you’ve gotten this far into reading chapter 1, I suspect that

▪ You’re a Java developer.

▪ You have a background in Spring.

▪ You’re interested in learning how to build microservice-based applications.

▪ You’re interested in how to use microservices to build cloud-based applications.

▪ You want to know if Java and Spring are relevant technologies for building microservice-based applications.

▪ You’re interested in seeing what goes into deploying a microservice-based appli- cation to the cloud.

I chose to write this book for two reasons. First, while I’ve seen many good books on the conceptual aspects of microservices, I couldn’t a find a good Java-based book on imple- menting microservices. While I’ve always considered myself a programming language polyglot (someone who knows and speaks several languages), Java is my core develop- ment language and Spring has been the development framework I “reach” for when- ever 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.

Second, as I’ve worked throughout my career as both an architect and engineer, I’ve found that many times the technology books that I purchase have tended to go to one of two extremes. They are either conceptual without concrete code examples, or

**Licensed to <null>**

**8** CHAPTER 1 ***Welcome to the cloud, Spring***

are mechanical overviews of a particular framework or programming language. I wanted a book that would be a good bridge and middle ground between the architec- ture and engineering disciplines. As you read this book, I want to give you a solid introduction to the microservice patterns development and how they’re used in real- world application development, and then back these patterns up with practical and easy-to-understand code examples using Spring Boot and Spring Cloud.

Let’s shift gears for a moment and walk through building a simple microservice using Spring Boot.

***1.5 Building a microservice with Spring Boot***

I’ve always had 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’s 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 see how to write a simple “Hello World” REST-service using Spring Boot.

In this section, we’re not going to do a detailed walkthrough of much of the code presented. Our goal is to give you a taste of writing a Spring Boot service. We’ll go into much more detail in chapter 2.

Figure 1.3 shows what your service is going to do and the general flow of how Spring Boot microservice will process a user’s request.

This example is by no means exhaustive or even illustrative of how you should build a production-level microservice, but it should cause you to take a pause because of how little code it took to write it. We’re not going to go through how to set up the project build files or the details of the code until chapter 2. If you’d 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. All the source code for chapter 1 can be retrieved from the GitHub repository for the book at https://github.com/carnellj/spmia-chapter1.

NOTE Please make sure you read appendix A before you try to run the code examples for the chapters in this book. Appendix A covers the general pro- ject layout of all the projects in the book, how to run the build scripts, and how to fire up the Docker environment. The code examples in this chapter are simple and designed to be run natively right from your desktop without the information in additional chapters. However, in later chapters you’ll quickly begin using Docker to run all the services and infrastructure used in this book. Don’t go too far into the book without reading appendix A on set- ting up your desktop environment.

**Licensed to <null>**

**9 *Building a microservice with Spring Boot***

GET http://localhost:8080/hello/john/carnell

**A client makes an HTTP GET request to your Hello microservice.** HTTP STATUS:200 {"message": "Hello john carnell"}

**The client receives the response from your service as JSON. The success or failure of** Route mapping **the call is returned as an HTTP status code.**

**Once Spring Boot has identified the route it will map any parameters defined inside the route to a Java method that will carry out the work.**

**Once all of the data has been mapped, Spring Boot will execute the business logic.**

Flow of Spring Boot microservice

For this example, you’re going to have a single Java class called simpleservice/ src/com/thoughtmechanix/application/simpleservice/Application.java that will be used to expose a REST endpoint called /hello.

The following listing shows the code for Application.java.

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;

**Spring Boot will parse the HTTP request and map the route based on the HTTP Verb, the URL, and potential parameters defined for the URL. A route maps to a method in a Spring** Parameter

**RestController class.** destructuring

**For an HTTP PUT or Post,** JSON->Java object mapping

**a JSON passed in the HTTP body is mapped to a Java class.**

Business logic execution **Once the business logic is executed, Spring Boot** Java->JSON object mapping

**will convert a Java object to JSON.**

Figure 1.3 Spring Boot abstracts away the common REST microservice task (routing to business logic, parsing HTTP parameters from the URL, mapping JSON to/from Java Objects), and lets the developer focus on the business logic for the service.

Listing 1.1 Hello World with Spring Boot: a simple Spring microservice

**Licensed to <null>**

**10** CHAPTER 1 ***Welcome to the cloud, Spring***

**Tells the Spring Boot framework that this class is the entry point for the Spring Boot service Tells Spring Boot you’re going to expose the code in this class** @SpringBootApplication **as a Spring RestController class**

@RestController @RequestMapping(value="hello") public class Application {

**All URLs exposed in this application will be prefaced with /hello prefix.**

public static void main(String[] args) {

SpringApplication.run(Application.class, args); }

**Spring Boot will expose an endpoint as a GET-based REST endpoint that will take two parameters: firstName and lastName.** @RequestMapping(value="/{firstName}/{lastName}", method = RequestMethod.GET) public String hello( @PathVariable("firstName") String firstName, @PathVariable("lastName") String lastName) { return String.format("{\"message\":\"Hello %s %s\"}",

firstName, lastName); } }**Maps the firstName and lastName Returns a simple JSON string that you manually build. In chapter 2 you won’t create any JSON.parameters passed in on the URL to two variables passed into the hello function** In listing 1.1 you’re 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 payload containing the message “Hello *firstName lastName*”. If you were to call the endpoint /hello/john/carnell on your service (which I’ll show shortly) the return of the call would be

{"message":"Hello john carnell"}

Let’s fire up your service. To do this, go to the command prompt and issue the follow- ing 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 program- ming languages. You can build microservices with Groovy and no project setup. Spring Boot also supports both Maven and the Gradle build tools. I’ve limited the examples in this book to Java and Maven. As a long-time Groovy and Gradle aficio- nado, I have a healthy respect for the language and the build tool, but to keep the book manageable and the material focused, I’ve chosen to go with Java and Maven to reach the largest audience possible.

**Licensed to <null>**

**11 *Building a microservice with Spring Boot***

**Our /hello endpoint is mapped with two variables: firstName and lastName.**

**The service will listen to port 8080 for incoming HTTP requests.**

Figure 1.4 Your Spring Boot service will communicate the endpoints exposed and the port of the service via the console.

If everything starts correctly, you should see what’s shown in figure 1.4 from your command-line window.

If you examine the screen in figure 1.4, you’ll 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.

**HTTP GET for the /hello/john/carnell endpoint**

**JSON payload returned back from the service**

Figure 1.5 The response from the **/hello** endpoint shows the data you’ve requested represented as a JSON payload.

**Licensed to <null>**

**12** CHAPTER 1 ***Welcome to the cloud, Spring***

You’re going to call your service using a browser-based REST tool called POSTMAN (https://www.getpostman.com/). Many tools, both graphical and command line, are available for invoking a REST-based service, but I’ll use POSTMAN for all my examples in this book. Figure 1.5 shows the POSTMAN call to the http://localhost:8080/ hello/john/carnell endpoint and the results returned from the service.

Obviously, this simple example doesn’t 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 few 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 powerful language, has acquired a reputation of being wordy compared to other languages.

We’re done with our brief tour of Spring Boot. We now have to ask this question: because we can write our applications using a microservice approach, does this mean we should? In the next section, we’ll walk through why and when a microservice approach is justified for building your applications.

***1.6 Why change the way we build applications?***

We’re 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 sud- denly finding that they can reach out to a global customer base. However, with a larger global customer base also comes global competition. These competitive pres- sures 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 and don’t integrate with other applications are no longer the norm. Today’s applications need to talk to multiple services and databases residing not only inside a com- pany’s data center, but also to external service providers over the internet.

▪ *Customers want faster delivery*—Customers no longer want to wait for the next annual 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 quickly in weeks (even days) 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 scale up across multiple servers quickly and then scale back down when the volume needs have passed.

▪ *Customers expect their applications to be available*—Because customers are one click away from a competitor, a company’s applications must be highly resilient. Fail- ures or problems in one part of the application shouldn’t bring down the entire application.

**Licensed to <null>**

**13 *What exactly is the cloud?***

To meet these expectations, we, as application developers, have to embrace the para- dox that to build high-scalable and highly redundant applications we need to break our applications into small services that can be built and deployed independently of one another. If we “unbundle” our applications into small services 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 mean an application is no longer a single “ball of mud” where a degradation in one part of the application causes the whole appli- cation to fail. Failures can be localized to a small part of the application and con- tained before the entire application experiences an outage. This also enables the applications to degrade gracefully in case of an unrecoverable error.

▪ *Scalable*—Decoupled services can easily be distributed horizontally across multi- ple servers, making it possible to scale the features/services appropriately. With a monolithic application where all 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 is 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 every- one’s platform is cloud-enabled, but if you cut through the hype, three basic models exist in 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 you want to eat a meal, you have four choices:

1 You can make the meal at home. 2 You can go to the grocery store and buy a meal pre-made that you heat up and

serve. 3 You can get a meal delivered to your house. 4 You can get in the car and eat at restaurant.

**Licensed to <null>**

**14** CHAPTER 1 ***Welcome to the cloud, Spring***

Furniture

Plates

Oven

Ingredients

Chef

Homemade

On premise

Furniture

Plates

Oven

Ingredients

Furniture

Furniture

Plates

Plates

Oven

Oven

Ingredients

Ingredients

Chef

Delivered

PaaS

Furniture

Furniture

Furniture

Plates

Plates

Plates

Oven

Oven

Oven

Ingredients

Ingredients

Ingredients

Chef

Chef

Chef

Chef

Chef

Store bought

Store bought

Restaurant

Restaurant

Restaurant

IaaS

IaaS

SaaS

SaaS

SaaS

Figure 1.6 The different cloud computing models come down to who’s responsible for what: the cloud vendor or you.

Figure 1.6 shows each model.

The difference between these options is 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 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. You’re using the store’s chef and oven to pre-bake the meal, but you’re still responsible for heating the meal and eating it at the house (and cleaning up the dishes afterward).

In a Platform as a Service (PaaS) model you still have responsibility for the meal, but you further rely on a vendor to take care of the core tasks associated with making a meal. For example, in a PaaS model, you supply the plates and furniture, but the res- taurant owner provides the oven, ingredients, and the chef to cook them. In the Soft- ware as a Service (SaaS) model, you go to a restaurant where all the food is prepared for you. You eat at the restaurant and then you pay for the meal when you’re done. you also have no dishes to prepare or wash.

The key items at play in each of these models are ones of control: who’s responsi- ble 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’re accountable for selecting the technology and building the final solution. On the other end of the spectrum, with a SaaS model, you’re 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.

You manage Provider manages

**Licensed to <null>**

**15 *Why the cloud and microservices?***

Emerging cloud platforms I’ve documented the three core cloud platform types (IaaS, PaaS, SaaS) that are in use today. However, new cloud platform types are emerging. These new platforms include Functions as a Service (FaaS) and Container as a Service (CaaS). FaaS-based (https://en.wikipedia.org/wiki/Function\_as\_a\_Service) applications use technolo- gies like Amazon’s Lambda technologies and Google Cloud functions to build appli- cations deployed as “serverless” chunks of code that run completely on the cloud provider’s platform computing infrastructure. With a FaaS platform, you don’t have to manage any server infrastructure and only pay for the computing cycles required to execute the function.

With the Container as a Service (CaaS) model, developers build and deploy their microservices as portable virtual containers (such as Docker) to a cloud provider. Unlike an IaaS model, where you the developer have to manage the virtual machine the service is deployed to, with CaaS you’re deploying your services in a lightweight virtual container. The cloud provider runs the virtual server the container is running on as well as the provider’s comprehensive tools for building, deploying, monitoring, and scaling containers. Amazon’s Elastic Container Service (ECS) is an example of a CaaS-based platform. In chapter 10 of this book, we’ll see how to deploy the microservices you’ve built to Amazon ECS.

It’s important to note that with both the FaaS and CaaS models of cloud computing, you can still build a microservice-based architecture. Remember, the concept of microservices revolves around building small services, with limited responsibility, using an HTTP-based interface to communicate. The emerging cloud computing plat- forms, such as FaaS and CaaS, are really about alternative infrastructure mecha- nisms for deploying microservices.

***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. Service instances should be brought up quickly and each instance of the service should be indistin- guishable from another.

As a developer writing a microservice, sooner or later you’re going to have to decide whether your service is going to be deployed to one of the following:

▪ *Physical server*—While you can build and deploy your microservices to a physi- cal machine(s), few organizations do this because physical servers are con- strained. 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 their ability to quickly start up and shut down microservice instances in response to scalabil- ity and service failure events. Virtual machines are the heart and soul of the

**Licensed to <null>**

**16** CHAPTER 1 ***Welcome to the cloud, Spring***

major cloud providers. A microservice can be packaged up in a virtual machine image and 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 of 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 contain- ers (or equivalent container technology) to the cloud. Virtual containers run inside a virtual machine; using a virtual container, you can segregate a single vir- tual machine into a series of self-contained processes that share the same virtual machine image.

The advantage of cloud-based microservices centers around the concept of elasticity. Cloud service providers allow you to quickly spin up new virtual machines and contain- ers in a matter of minutes. If your capacity needs for your services drop, you can spin down virtual servers without incurring any additional costs. Using a cloud provider to deploy your microservices gives you significantly more horizontal scalability (adding more servers and service instances) for your applications. Server elasticity also means that your applications can be more resilient. If one of your microservices is having prob- lems and is falling over, spinning up new service instances can you keep your applica- tion alive long enough for your development team to gracefully resolve the issue.

For this book, all the microservices and corresponding service infrastructure will be deployed to an IaaS-based cloud provider using Docker containers. This is a com- mon 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 suc- cinctly start one or more instances of a service. This capability means you can quickly scale services and 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 using clusters in a data center.

**Licensed to <null>**

**17 *Microservices are more than writing the code***

Why not PaaS-based microservices? Earlier in the chapter we discussed three types of cloud platforms (Infrastructure as a Service, Platform as a Service, and Software as a Services). For this book, I’ve cho- sen to focus specifically on building microservices using an IaaS-based approach. While certain cloud providers will let you abstract away the deployment infrastructure for your microservice, I’ve chosen to remain vendor-independent and deploy all parts of my application (including the servers).

For instance, Amazon, Cloud Foundry, and Heroku give you the ability to deploy your services without having to know about the underlying application container. They pro- vide 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 con- tainer are abstracted away from you. While this is convenient, each cloud provider’s platform has different idiosyncrasies related to its individual PaaS solution.

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

Earlier in the chapter, I mentioned new cloud computing platforms such as Function as a Service (FaaS) and Container as a Service (CaaS). If you’re not careful, FaaS- based platforms can lock your code into a cloud vendor platform because your code is deployed to a vendor-specific runtime engine. With a FaaS-based model, you might be writing your service using a general programming language (Java, Python, JavaS- cript, and so on), but you’re still tying yourself heavily to the underlying vendor APIs and runtime engine that your function will be deployed to.

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 the major cloud providers. Later in chapter 10, I demonstrate how to package microser- vices using Docker and then deploy these containers to Amazon’s cloud platform.

***1.9 Microservices are more than writing the code***

While the concepts around building individual microservices are easy to understand, running and supporting a robust microservice application (especially when running

**Licensed to <null>**

**18** CHAPTER 1 ***Welcome to the cloud, Spring***

**How do you manage the physical location so services instances can be added and removed without impacting service clients?**

**How do you make sure the service is focused** Location

transparent **on one area of responsibility?**

Your microservice Resilient

Scalable Repeatable

Figure 1.7 Microservices are more than the business logic. You need to think about the environment where the services are going to run and how the services will scale and be resilient.

in the cloud) involves more than writing the code for the service. Writing a robust ser- vice includes considering several topics. Figure 1.7 highlights these topics.

Let’s walk through the items in figure 1.7 in more detail:

▪ *Right-sized*—How do you ensure that your microservices are properly sized so that you don’t have a microservice take on too much responsibility? Remember, properly sized, a service allows you to quickly make changes to an application and reduces the overall risk of an outage to the entire application.

▪ *Location transparent—*How you we manage the physical details of service invoca- tion when in a microservice application, multiple service instances can quickly start and shut down?

▪ *Resilient*—How do you protect your microservice consumers and the overall integrity of your application by routing around failing services and ensuring that you take a “fail-fast” approach?

▪ *Repeatable*—How do you ensure that every new instance of your service brought up is guaranteed to have the same configuration and code base as all the other service instances in production?

▪ *Scalable*—How do you use asynchronous processing and events to minimize the direct dependencies between your services and ensure that you can gracefully scale your microservices?

This book takes a patterns-based approach as we answer these questions. With a pat- terns-based approach, we lay out common designs that can be used across different

**Licensed to <null>**

**How do you make sure when there is a problem with a service, service clients “fail fast”?**

Right-sized

**How do you ensure that your applications can scale quickly with minimal dependencies between services?**

**How do you ensure that every time a new service instance is started it always has the same code and configuration as existing instance(s)?**

**19 *Microservices are more than writing the code***

technology implementations. While we’ve chosen to use Spring Boot and Spring Cloud to implement the patterns we’re going to use in this book, nothing will keep you from taking the concepts presented here and using them with other technology platforms. Specifically, we cover the following six categories of microservice patterns:

▪ Core development patterns

▪ Routing patterns

▪ Client resiliency patterns

▪ Security patterns

▪ Logging and tracing patterns

▪ Build and deployment patterns

Let’s walk through these patterns in more detail.

***1.9.1 Core microservice development pattern***

The core development microservice development pattern addresses the basics of building a microservice. Figure 1.8 highlights the topics we’ll cover around basic ser- vice design.

Figure 1.8 When designing your microservice, you have to think about how the service will be consumed and communicated with.

Web client Microservice

Microservice

Communication protocols **Service granularity: What is the right level of responsibility the** Service

**service should have?** granularity

**Interface design: How you are going to expose your service** Interface

**endpoints to clients** design

Configuration management

**Event processing: How you can use events to communicate** Event processing **Communication protocols: How your client and service communicate data back and forth**

**Configuration management: How your services manage their application-specific configuration so that the code and configuration are independent entities**

**state and data changes between services**

**Licensed to <null>**

**20** CHAPTER 1 ***Welcome to the cloud, Spring***

▪ *Service granularity*—How do you approach decomposing a business domain down into microservices so that each microservice has the right level of respon- sibility? Making a service too coarse-grained with responsibilities that overlap into different business problems domains makes the service difficult to main- tain and change over time. Making the service too fine-grained increases the overall complexity of the application and turns the service into a “dumb” data abstraction layer with no logic except for that needed to access the data store. I cover service granularity in chapter 2.

▪ *Communication protocols*—How will developers communicate with your service? Do you use XML (Extensible Markup Language), JSON (JavaScript Object Nota- tion), or a binary protocol such as Thrift to send data back and forth your microservices? 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 microservices. I cover communication protocols in chapter 2.

▪ *Interface design*—What’s the best way to design the actual service interfaces that developers are going to use to call your service? How do you structure your ser- vice 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 8.

***1.9.2 Microservice routing patterns***

The microservice routing patterns deal with how a client application that wants to consume a microservice discovers the location of the service and is routed over to it. In a cloud-based application, you might have hundreds of microservice instances run- ning. You’ll 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 discovery and routing answer the question, “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 hard- coded into the application? How do you ensure that misbehaving microservice instances are removed from the pool of available service instances? I cover ser- vice discovery in chapter 4.

**Licensed to <null>**

**21 *Microservices are more than writing the code***

Web client

Microservice

http://myapp.api/servicea http://myapp.api/serviceb

**Service routing gives the microservice client a single logical URL to talk to and acts as a policy enforcement point for things like authorization, authentication, and content checking.**

**Service discovery abstracts away the physical location of the service from the client. New microservice instances can be added to scale up, and unhealthy service instances can be transparently removed from the service.**

Figure 1.9 Service discovery and routing are key parts of any large-scale microservice application.

▪ *Service routing*—How do you 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 your 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 1.9, service discovery and service routing appear to have a hard-coded sequence of events between them (first comes service routing and the service discov- ery). However, the two patterns aren’t dependent on one another. 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***

Because microservice architectures are highly distributed, you have to be extremely sensitive in how you prevent a problem in a single service (or service instance) from

**Licensed to <null>**

172.18.32.100 172.18.32.101

Microservice A (two instances)

172.18.38.96 172.18.38.97

Microservice B (two instances)

**22** CHAPTER 1 ***Welcome to the cloud, Spring***

cascading up and out to the consumers of the service. To this end, we’ll cover four cli- ent resiliency patterns:

▪ *Client-side load balancing*—How do you cache the location of your service instances on the service client so that calls to multiple instances of a microser- vice are load balanced to all the health instances of that microservice?

▪ *Circuit breakers pattern*—How do you prevent a client from continuing to call a service that’s failing or suffering performance problems? When a service is run- ning slowly, it consumes resources on the client calling it. You want failing microservice calls to fail fast so that the calling client can quickly respond and take an appropriate action.

▪ *Fallback pattern*—When a service call fails, how do you provide a “plug-in” mech- anism that will allow the service client to try to carry out its work through alter- native means other than the microservice being called?

▪ *Bulkhead pattern—*Microservice applications use multiple distributed resources to carry out their work. How do you compartmentalize these calls so that the mis- behavior of one service call doesn’t negatively impact the rest of the application?

Web client

Microservice

http://myapp.api/servicea http://myapp.api/serviceb

**The circuit breaker pattern ensures that a service client** Figure 1.10 With microservices, you must protect the service caller from a poorly behaving service. Remember, a slow or down service can cause disruptions beyond the immediate service.

Client-side load balancing **does not repeatedly call a failing service. Instead, a circuit breaker "fails fast" to protect the client.**

Circuit breaker

**How do you segregate different** Fallback **service calls on a client to make sure one misbehaving service does not take up all the resources on the client?**

Bulkhead

**Licensed to <null>**

**The service client caches microservice endpoints retrieved from the service discovery and ensures that the service calls are load balanced between instances.**

172.18.32.101

Microservice A (two instances)

**When a client does fail, is there an alternative path the client can take to retrieve data from or take action with?**

172.18.32.100

172.18.38.96 172.18.38.97

Microservice B (two instances)

**23 *Microservices are more than writing the code***

Figure 1.10 shows how these patterns protect the consumer of service from being impacted when a service is misbehaving. I cover these four topics in chapter 5.

***1.9.4 Microservice security patterns***

I can’t write a book on microservices without talking about microservice security. In chapter 7 we’ll cover three basic security patterns. These patterns are

▪ *Authentication*—How do you determine the service client calling the service is who they say they are?

▪ *Authorization*—How do you determine whether the service client calling a microservice is allowed to undertake the action they’re trying to undertake?

▪ *Credential management and propagation*—How do you prevent a service client from constantly having to present their credentials for service calls involved in a trans- action? Specifically, we’ll look at how token-based security standards such as OAuth2 and JavaScript Web Tokens (JWT) can be used to obtain a token that can be passed from service call to service call to authenticate and authorize the user.

Figure 1.11 shows how you can implement the three patterns described previously to build an authentication service that can protect your microservices.

At this point I’m not going to go too deeply into the details of figure 1.10. There’s a reason why security requires a whole chapter. (It could honestly be a book in itself.)

**1. The service you 4. The token server authenticates want to protect**

**the user and validates tokens presented to it**

Token authentication server

Protected resource

Application trying to

The user

access a protected resource

Resource owner

Figure 1.11 Using a token-based security scheme, you can implement service authentication and authorization without passing around client credentials.

**Licensed to <null>**

**3. When the user tries to access a protected service, they must authenticate and obtain a token from the authentication service. 2. The resource owner grants which applications/users can access the resource via the authentication service**

**24** CHAPTER 1 ***Welcome to the cloud, Spring***

***1.9.5 Microservice logging and tracing patterns***

The beauty of the microservice architecture is that a monolithic application is broken down into small pieces of functionality that can be deployed independently of one another. The downside of a microservice architecture is that it’s much more difficult to debug and trace what the heck is going on within your application and services.

For this reason, we’ll look at three core logging and tracing patterns:

▪ *Log correlation*—How do you tie together all the logs produced between services for a single user transaction? With this pattern, we’ll look at how to implement a correlation ID, which is a unique identifier that will be carried across all service calls in a transaction and can be used to tie together log entries produced from each service.

▪ *Log aggregation*—With this pattern we’ll look at how to pull together all of the logs produced by your microservices (and their individual instances) into a sin- gle queryable database. We’ll also look at how to use correlation IDs to assist in searching your aggregated logs.

▪ *Microservice tracing*—Finally, we’ll explore how to visualize the flow of a client transaction across all the services involved and understand the performance characteristics of services involved in the transaction.

Figure 1.12 shows how these patterns fit together. We’ll cover the logging and tracing patterns in greater detail in chapter 9.

Service instance A Service instance A Service instance B Service instance B Service instance C

**Log correlation: All service log entries have a correlation ID that ties the log entry to a single transaction.**

**Log aggegration: An aggregation mechanism collects all of the logs from all the services instances.**

Figure 1.12 A well-thought-out logging and tracing strategy makes debugging transactions across multiple services manageable.

**Licensed to <null>**

**As data comes into a central data store, it is indexed and stored in a searchable format.**

**Microservice transaction tracing: The development and operations teams can query the log data to find individual transactions. They should also be able to visualize the flow of all the services involved in a transaction.**

**25 *Microservices are more than writing the code***

***1.9.6 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 its other instances. You can’t allow “configura- tion drift” (something changes on a server after it’s been deployed) to occur, because 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’ve worked on critical sit- uations 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’re doing the best they can, but they get busy or distracted. They tweak some- thing on a server, fully intending to go back and do it in all the environments. At a later point, an outage occurs and everyone is left scratching their heads wondering what’s different between the lower environments in production. I’ve found that the small size and limited scope of a microservice makes it the perfect opportunity to intro- duce the concept of “immutable infrastructure” into an organization: once a service is deployed, the infrastructure it’s 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 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 your infrastructure right into your build-deployment process so that you no longer deploy software artifacts such as a Java WAR or EAR to an already-running piece of infrastructure. Instead, you want to build and compile your microservice and the virtual server image it’s running on as part of the build process. Then, when your microservice gets deployed, the entire machine image with the server running on it gets deployed.

Figure 1.13 illustrates this process. At the end of the book we’ll look at how to change your build and deployment pipeline so that your microservices and the servers they run on are deployed as a single unit of work. In chapter 10 we cover the following patterns and topics:

▪ *Build and deployment pipeline—*How do you create a repeatable build and deploy- ment process that emphasizes one-button builds and deployment to any envi- ronment 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’s never changed after it has been deployed?

▪ *Phoenix servers—*The longer a server is running, the more opportunity for con- figuration drift. How do you ensure that servers that run microservices get torn down on a regular basis and recreated off an immutable image?

**Licensed to <null>**

**26** CHAPTER 1 ***Welcome to the cloud, Spring***

**Everything starts with a developer checking in their code to a source control repository. This is the trigger to begin the build/deployment process.**

Continuous integration/continuous delivery pipeline

Code compiled

Developer Source repository **Infrastructure as code: We build our code and run our tests for our microservices. However, we also treat our infrastructure as code. When the microservice is compiled and packaged, we immediately bake and provision a virtual server or container image with the microservice installed on it.**

**Immutable servers: The moment an image is baked and deployed, no developer or system administrator is allowed to make modifications to the servers. When promoting between environments, the entire container or image is started with environment-specific variables that are passed to the server when the server is first started.**

**Phoenix servers: Because the actual servers are constantly being torn down as part of the continous integration process, new servers are being started and torn down. This greatly decreases the change of configuration drift between environments.**

Figure 1.13 You want the deployment of the microservice and the server it’s running on to be one atomic artifact that’s deployed as a whole between environments.

Our goal with these patterns and topics is to ruthlessly expose and stamp out configu- ration drift as quickly as possible before it can hit your upper environments, such as stage or production.

NOTE For the code examples in this book (except chapter 10), everything will run locally on your desktop machine. The first two chapters can be run natively directly from the command line. Starting in chapter 3, all the code will be compiled and run as Docker containers.

***1.10 Using Spring Cloud in building your microservices***

In this section, I briefly introduce the Spring Cloud technologies that you’ll use as you build out your microservices. This is a high-level overview; when you use each technol- ogy in this book, I’ll teach you the details on each as needed.

Unit and

Run-time integration tests run

Machine artifacts

image created

baked

Platform test run

Image deploy/new server deployed

Platform test run

Image deploy/new server deployed

Platform test run

Image committed

Build deploy

to repo

engine

Dev

Test

Prod

Image deploy/new server deployed

**Licensed to <null>**

**27 *Using Spring Cloud in building your microservices***

Implementing all these patterns 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 subproject collectively known as Spring Cloud. (http://projects.spring.io/spring-cloud/).

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

Figure 1.14 maps the patterns listed in the previous section to the Spring Cloud projects that implement them.

Let’s walk through these technologies in greater detail.

Development patterns

Routing patterns Client resiliency patterns

Build deployment patterns Client-side load balancing

Continuous

Core microservice

Spring Cloud/

integration

patterns

Netflix Ribbon

Travis CI

Spring Boot

Service discovery patterns

Circuit breaker pattern

Infrastructure

Spring Cloud/

Spring Cloud/

as code

Configuration

Netflix Eureka

Netflix Hystrix

Docker management

Spring Cloud Config

Service routing

Fallback pattern

patterns

Spring Cloud/

Immutable servers

Asynchronous

Spring Cloud/

Netflix Hystrix

Docker

messaging

Netflix Zuul

Spring Cloud Stream

Bulkhead pattern

Phoenix servers

Spring Cloud/

Travis CI/Docker Netflix Hystrix

Logging patterns

Log correlation

Log aggregation

Microservice tracing

Spring Cloud Sleuth

Spring Cloud Sleuth (with Papertrail)

Spring Cloud Sleuth/Zipkin

Security patterns

Authorization

Authentication

Credential management and propagation Spring Cloud Security/OAuth2

Spring Cloud Security/OAuth2

Spring Cloud Security/OAuth2/JWT

Figure 1.14 You can map the technologies you’re going to use directly to the microservice patterns we’ve explored so far in this chapter.

**Licensed to <null>**

**28** CHAPTER 1 ***Welcome to the cloud, Spring***

***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 also 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, as well as 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’ll always have the same configuration. Spring Cloud Config has its own property management repository, but also integrates with open source projects such as the following:

▪ *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 application’s 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 Con- fig to store application configuration data.

▪ *Eureka*—Eureka (https://github.com/Netflix/eureka) is an open source Net- flix 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 invoke 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’re started up and shut down. Spring Cloud service discovery can be implemented using Consul (https:// www.consul.io/) and Eureka (https://github.com/Netflix/eureka) as its service dis- covery engine.

**Licensed to <null>**

**29 *Using Spring Cloud in building your microservices***

***1.10.4 Spring Cloud/Netflix Hystrix and Ribbon***

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

Using the Netflix Hystrix libraries, you can quickly implement service client resil- iency patterns such as the circuit breaker and bulkhead patterns.

While the Netflix Ribbon project simplifies integrating with service discovery agents such as 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 uses the Netflix Zuul project (https://github.com/Netflix/zuul) to pro- vide service routing capabilities for your microservice application. Zuul is a service gateway that proxies service requests and makes sure that all calls to your microser- vices go through a single “front door” before the targeted service is invoked. With this centralization of service calls, you can enforce standard service policies such as a secu- rity 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 to easily integrate lightweight message processing into your microservice. Using Spring Cloud Stream, you can build intelligent microservices that can use asynchronous events as they occur in your application. With Spring Cloud Stream, you can quickly integrate your microservices with message brokers such as RabbitMQ (https://www.rabbitmq.com/) and Kafka (http://kafka.apache.org/).

***1.10.7 Spring Cloud Sleuth***

Spring Cloud Sleuth (https://cloud.spring.io/spring-cloud-sleuth/) allows you to integrate unique tracking identifiers into the HTTP calls and message channels (Rab- bitMQ, Apache Kafka) being used within your application. These tracking numbers, sometimes referred to as correlation or trace ids, allow you to track a transaction as it flows across the different services in your application. With Spring Cloud Sleuth, these trace IDs are automatically added to any logging statements you make in your microservice.

The real beauty of Spring Cloud Sleuth is seen when it’s combined with logging aggregation technology tools such as Papertrail (http://papertrailapp.com) and trac- ing tools such as Zipkin (http://zipkin.io). Papertail is a cloud-based logging platform used to aggregate logs in real time from different microservices into one queryable

**Licensed to <null>**

**30** CHAPTER 1 ***Welcome to the cloud, Spring***

database. Open Zipkin takes data produced by Spring Cloud Sleuth and allows you to visualize the flow of your service calls involved for a single transaction.

***1.10.8 Spring Cloud Security***

Spring Cloud Security (https://cloud.spring.io/spring-cloud-security/) is an authenti- cation and authorization framework that can control who can access your services and what they can do with your services. Spring Cloud Security is token-based and allows services to communicate with one another through a token issued by an authentica- tion server. Each service receiving a call can check the provided token in the HTTP call to validate the user’s identity and their access rights with the service.

In addition, Spring Cloud Security supports the JavaScript Web Token (https:// jwt.io). The JavaScript Web Token (JWT) framework standardizes the format of how a OAuth2 token is created and provides standards for digitally signing a created token.

***1.10.9 What about provisioning?***

For the provisioning implementations, we’re going to make a technology shift. The Spring framework(s) are geared toward application development. The Spring frame- works (including Spring Cloud) don’t have tools for creating a “build and deployment” pipeline. To implement a “build and deployment” pipeline you’re going to use the fol- lowing tools: Travis CI (https://travis-ci.org) for your build tool and Docker (https:// www.docker.com/) to build the final server image containing your microservice.

To deploy your built Docker containers, we end the book with an example of how to deploy the entire application stack built throughout this book to Amazon’s cloud.

***1.11 Spring Cloud by example***

In the last section, we walked through all the different Spring Cloud technologies that you’re going to use as you build out your microservices. Because each of these tech- nologies are independent services, it’s 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 into your own microservice development effort.

Unlike the first code example in listing 1.1, you can’t run this code example because a number of supporting services need to be set up and configured to be used. 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’re set up, your individual microservices can use these capabilities over and over again. We couldn’t fit all that goodness into a single code example at the beginning of the book. The code shown in the following listing quickly demonstrates how the service dis- covery, circuit breaker, bulkhead, and client-side load balancing of remote services were integrated into our “Hello World” example.

**Licensed to <null>**

**31 *Spring Cloud by example***

Listing 1.2 Hello World Service using 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

**Enables the service to use the Hystrix and Ribbon libraries** @EnableEurekaClient public class Application {

**Tells the service that it should register itself with a Eureka** public static void main(String[] args) { SpringApplication.run(Application.class, }

**service discovery agent and that** args);

**service calls are to use service discovery to “lookup” the location of remote services**

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

String lastName){ ResponseEntity<String> restExchange =

restTemplate.exchange(

"http://**logical-service-id**/name/ **Wrappers calls to the helloRemoteServiceCall** [ca]{firstName}/{lastName}", **method with a Hystrix** HttpMethod.GET, **circuit breaker** null, String.class, firstName, lastName);

return restExchange.getBody(); **Uses a decorated RestTemplate class to take a “logical” service ID and Eureka under the covers to look up the physical location** }

**of the service**

@RequestMapping(value="/{firstName}/{lastName}",

method = RequestMethod.GET) public String hello( @PathVariable("firstName") String firstName, @PathVariable("lastName") String lastName) { return helloRemoteServiceCall(firstName, lastName) }}This code has a lot packed into it, so let’s walk through it. Keep in mind that this list- ing is only an example and isn’t found in the chapter 1 GitHub repository source code. I’ve included it here to give you a taste of what’s to come later in the book.

The first thing you should notice is the @EnableCircuitBreaker and @EnableEurekaClient annotations. The @EnableCircuitBreaker annotation tells your Spring microservice that you’re going to use the Netflix Hystrix libraries in your application. The @EnableEurekaClient annotation tells your microservice to

**Licensed to <null>**

**32** CHAPTER 1 ***Welcome to the cloud, Spring***

register itself with a Eureka Service Discovery agent and that you’re going to use service discovery to look up remote REST services endpoints in your code. Note that configuration is happening in a property file that will tell the simple service the loca- tion and port number of a Eureka server to contact. You first see Hystrix being used when you declare your hello method:

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

The @HystrixCommand annotation is doing two things. First, any time the helloRe- moteServiceCall method is called, it won’t be directly invoked. Instead, the method will be delegated to a thread pool managed by Hystrix. If the call takes too long (default is one second), Hystrix steps in and interrupts the call. This is the imple- mentation of the circuit breaker pattern. The second thing this annotation does is cre- ate a thread pool called helloThreadPool that’s 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’s occurring inside the helloRemoteServiceCall method. The presence of the @EnableEurekaClient has told Spring Boot that you’re going to use a modified RestTemplate class (this isn’t how the Standard Spring RestTemplate would work out of the box) whenever you make a REST service call. This RestTemplate class will allow you to pass in a logical service ID for the ser- vice you’re trying to invoke:

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

Under the covers, the RestTemplate class will contact the Eureka service and look up the physical location of one or more of the “name” service instances. As a con- sumer of the service, your code never has to know where that service is located.

Also, the RestTemplate class is using Netflix’s Ribbon library. Ribbon will retrieve a list of all the physical endpoints associated with a service. Every time the service is called by the client, it “round-robins” the call to the different service instances on the client without having to go through a centralized load balancer. By eliminating a cen- tralized load balancer and moving it to the client, you eliminate another failure point (load balancer going down) in your application infrastructure.

I hope that at this point you’re impressed, because you’ve added a significant num- ber of capabilities to your microservice with only a few annotations. That’s the real beauty behind Spring Cloud. You as a developer get to take advantage of battle-hard- ened microservice capabilities from premier cloud companies like Netflix and Con- sul. These capabilities, if used outside of Spring Cloud, can be complex and obtuse to set up. Spring Cloud simplifies their use to literally nothing more than a few simple Spring Cloud annotations and configuration entries.

**Licensed to <null>**

**33 *Summary***

***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’ve structured the chapters in this book and the cor- responding code examples around the adventures (misadventures) of a fictitious com- pany called ThoughtMechanix.

ThoughtMechanix is a software development company whose core product, Eagle- Eye, provides an enterprise-grade software asset management application. It provides coverage for all the critical elements: inventory, software delivery, license manage- ment, compliance, cost, and resource management. Its primary goal is to enable orga- nizations to gain an accurate point-in-time picture of its software assets.

The company is approximately 10 years old. While they’ve experienced solid reve- nue growth, internally they’re debating whether they should be re-platforming their core product from a monolithic on-premise-based application or move their applica- tion 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 archi- tecture. 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 examples in this book won’t build the entire ThoughtMechanix application. Instead you’ll build specific microservices from the problem domain at hand and then build the infrastructure that will support these services using various Spring Cloud (and some non-Spring-Cloud) technologies.

The ability to successfully adopt cloud-based, microservice architecture will impact all parts of a technical organization. This includes the architecture, engineering, test- ing, and operations teams. Input will be needed from each group and, in the end, they’re probably going to need reorganization as the team reevaluates their responsi- bilities in this new environment. Let’s start our journey with ThoughtMechanix as you begin the fundamental work of identifying and building out several of the microser- vices used in EagleEye 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.

▪ No industry standards exist for 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 microser- vice development patterns, including core development, routing patterns, cli- ent resiliency, security, logging, and build/deployment patterns.

**Licensed to <null>**

**34** CHAPTER 1 ***Welcome to the cloud, Spring***

▪ While microservices are language-agnostic, we introduced two Spring frame- works 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 such as Netflix and HashiCorp that have been “wrapped” with Spring annotations to significantly simplify the setup and configuration of these services.

**Licensed to <null>**

*Building microservices with Spring Boot*

***This chapter covers***

▪ Learning the key characteristics of a microservice

▪ Understanding how microservices fit into a cloud architecture

▪ Decomposing a business domain into a set of microservices

▪ Implementing a simple microservice using Spring Boot

▪ Understanding the perspectives for building microservice-based applications

▪ Learning when not to use microservices

The history of software development is littered with the tales of large development projects that after an investment of millions of dollars and hundreds of thousands of software developer hours, and with many 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 their own complexity and weight.

These mammoth projects tended to follow large, traditional waterfall develop- ment methodologies that insisted that all the application’s requirements and design be defined at the beginning of the project. So much emphasis was placed on

**35**

**Licensed to <null>**

**36** CHAPTER 2 ***Building microservices with Spring Boot***

getting all the specifications for the software “correct” that there was 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 isn’t a linear process of definition and execution, but rather an evolutionary one where it takes several iterations of *com- municating with*, *learning from*, *and delivering to* the customer before the development team truly 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 projects are

▪ *Tightly coupled—*The invocation of business logic happens at the programming- language level instead of through implementation-neutral protocols such as SOAP and REST. This greatly increases the chance that even a small change to an application component can break other pieces of the application and intro- duce new bugs.

▪ *Leaky—*Most large software applications manage different types of data. For instance, a customer relationship management (CRM) application might man- age customer, sales, and product information. In a traditional model, this data is kept in the same data model and within the same data store. Even though there are obvious boundaries between the data, too often it’s tempting for a team from one domain to directly access the data that belongs to another team. This easy access to data creates hidden dependencies and allows implemen- tation details of one component’s internal data structures to leak through the entire application. Even small changes to a single database table can require a significant number of code changes and regression-testing throughout the entire application.

▪ *Monolithic—*Because most of the application components for a traditional appli- cation reside in a single code base that’s shared across multiple teams, any time a change to the code is made, the entire application has to be recompiled, rerun through an entire testing cycle, and redeployed. Even small changes to the application’s code base, whether they’re new customer requirements or bug fixes, become expensive and time-consuming, and large changes become nearly impossible to do in a timely fashion.

A microservice-based architecture takes a different approach to delivering functional- ity. Specifically, microservice-based architectures have these characteristics:

▪ *Constrained—*Microservices have a single set of responsibilities and are narrow in scope. Microservices embrace the UNIX philosophy that an application is nothing more than a collection of services where each service does one thing and does that one thing really well.

▪ *Loosely coupled—A* microservice-based application is a collection of small ser- vices that only interact with one another through a non–implementation spe- cific interface using a non-proprietary invocation protocol (for example, HTTP

**Licensed to <null>**

**37**

and REST). As long as the interface for the service doesn’t change, the owners of the microservice have more freedom to make modifications to the service than 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 a more heavily interdependent, monolithic application.

Why are these microservice architecture attributes important to cloud-based develop- ment? Cloud-based applications in general have the following:

▪ *A large and diverse user base—*Different customers want different features, and they don’t want to have to wait for a long application release cycle before they can start using these features. Microservices allow features to be delivered quickly, because each service is small in scope and accessed through a well- defined interface.

▪ *Extremely high uptime 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 resistent to problems.

▪ *Uneven volume requirements—*Traditional applications deployed within the four walls of a corporate data center usually have consistent usage patterns that emerge over time. This makes capacity planning for these types of applications simple. But 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.

Because microservice applications are broken down into small components that can be deployed independently of one another, it’s 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 with 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 microser- vices as if you’re a police detective interviewing witnesses to a crime. Even though every witness saw the same events take place, their interpretation of the crime is shaped by their background, what was important to them (for example, what moti- vates them), and what environmental pressures were brought to bear at that moment

**Licensed to <null>**

**38** CHAPTER 2 ***Building microservices with Spring Boot***

they witnessed the event. Participants each have their own perspectives (and biases) of what they consider important.

Like a successful police detective trying to get to the truth, the journey to build a suc- cessful microservice architecture involves incorporating the perspectives of multiple individuals within your software development organization. Although it takes more than technical people to deliver an entire application, I believe that the foundation for successful microservice development starts with the perspectives of three critical roles:

▪ *The architect—*The architect’s job is to see the big picture and understand how an application can be decomposed into individual microservices and how the microservices will interact to deliver a solution.

▪ *The software developer—*The software developer 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 DevOps engineer brings intelligence to how the ser- vices are deployed and managed throughout not only production, but also all the nonproduction environments. The watchwords for the DevOps engineer are *consistency* and *repeatability* in every environment.

In this chapter, I’ll demonstrate how to design and build a set of microservices from the perspective of each of these roles using Spring Boot and Java. By the time the chapter concludes, you’ll 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 needs to be solved. The job of the architect is to provide the scaffolding against which developers will build their code so that all the pieces of the application fit together. When building a microservices architecture, a project’s architect focuses 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 try to break the problem on which they’re working into manageable chunks. They do this so they don’t have to try to fit all the details of the problem in their heads. Instead, they break the problem down abstractly into a few key parts and then look for the relationships that exist between these parts. In a microservices architecture, the architect breaks the business problem into chunks that represent discrete domains of activity. These chunks encapsulate the busi- ness rules and the data logic associated with a particular part of the business domain.

**Licensed to <null>**

**39 *The architect’s story: designing the microservice architecture***

Although you want microservices to encapsulate all the business rules for carrying out a single transaction, this isn’t always feasible. You’ll often have situations where you need to have groups of microservices working across different parts of the busi- ness domain to complete an entire transaction. An architect teases apart the service boundaries of a set of microservices by looking at where the data domain doesn’t seem to fit together.

For example, an architect might look at a business flow that’s to be carried out by code and realize that they need both customer and product information. The pres- ence of two discrete data domains is a good indication that multiple microservices are at play. How the two different parts of the business transaction interact usually becomes the service interface for the microservices.

Breaking apart a business domain is an art form rather than a black-and-white sci- ence. Use the following guidelines for identifying and decomposing a business prob- lem into microservice candidates:

1 *Describe the business problem, and listen to the nouns you’re using to describe the problem.* Using the same nouns over and over in describing the problem is usually an indication of a core business domain and an opportunity for a microservice. Examples of target nouns for the EagleEye domain from chapter 1 might look something like *contracts*, *licenses*, and *assets*. 2 *Pay attention to the verbs.* Verbs highlight actions and often represent the natural contours of a problem domain. If you find yourself saying “transaction X needs to get data from thing A and thing B,” that usually indicates that multiple ser- vices are at play. If you apply to EagleEye the approach of watching for verbs, you might look for statements such as, “When Mike from desktop services is setting up a new PC, he looks up the number of licenses available for software X and, if licenses are available, installs the software. He then updates the number of licenses used in his tracking spreadsheet.” The key verbs here are *looks* and *updates*. 3 *Look for data cohesion*. As you break apart your business problem into discrete pieces, look for pieces of data that are highly related to one another. If sud- denly, during the course of your conversation, you’re reading or updating data that’s radically different from what you’ve been discussing so far, you potentially have another service candidate. *Microservices should completely own their data.*

Let’s take these guidelines and apply them to a real-world problem. Chapter 1 intro- duced an existing software product called EagleEye that’s used for managing software assets such as software licenses and secure socket layer (SSL) certificates. These items are deployed to various servers throughout an organization.

EagleEye is a traditional monolithic web application that’s deployed to a J2EE application server residing within a customer’s data center. Your goal is to tease apart the existing monolithic application into a set of services.

**Licensed to <null>**

**40** CHAPTER 2 ***Building microservices with Spring Boot***

Rick (Procurement)

Figure 2.1 Interview the EagleEye users, and understand how they do their day-to-day work.

You’re going to start by interviewing all the users of the EagleEye application and dis- cussing with them how they interact and use EagleEye. Figure 2.1 captures a summary of the conversations you might have with the different business customers. By looking at how the users of EagleEye interact with the application and how the data model for the application is broken out, you can decompose the EagleEye problem domain into the following microservice candidates.

In the figure, I’ve highlighted a number of nouns and verbs that have come up during conversations with the business users. Because this is an existing application, you can look at the application and map the major nouns back to tables in the physi- cal data model. An existing application may have hundreds of tables, but each table will usually map back to a single set of logical entities.

Figure 2.2 shows a simplified data model based on conversations with EagleEye customers. Based on the business interviews and the data model, the microservice candidates are organization, license, contract, and assets services.

Ruth (Finance)

Mike (Desktop Services)

• Enters contract info into EagleEye

• Runs monthly cost reports

• Sets up PCs

• Defines types of software licenses

• Analyzes cost of licenses per

• Determines if software license for

• Enters how many licenses are

the contract

PC is available acquired with purchase

• Determines if licenses are over-

• Updates EagleEye with which or under-utilized

user has what software

• Cancels unused software licenses

EagleEye application

License

Contracts

Assets table

table

table

**EagleEye database: data model is shared and highly integrated.**

Organization License

Assets

Contract

Figure 2.2 A simplified EagleEye data model

**Licensed to <null>**

**41 *The architect’s story: designing the microservice architecture***

***2.1.2 Establishing service granularity***

Once you have a simplified data model, you can begin the process of defining what microservices you’re going to need in the application. Based on the data model in figure 2.2, you can see the potential for four microservices based on the following elements:

▪ Assets

▪ License

▪ Contract

▪ Organization The goal is to take these major pieces of functionality and extract them into com- pletely self-contained units that can be built and deployed independently of each other. But extracting services from the data model involves more than repackaging code into separate projects. It’s also about teasing out the actual database tables the services are accessing and only allowing each individual service to access the tables in its specific domain. Figure 2.3 shows how the application code and the data model become “chunked” into individual pieces.

**The EagleEye application is broken down from a monolithic** Figure 2.3 You use the data model as the basis for decomposing a monolithic application into microservices.

Assets tables

**application into smaller individual** License **services that are deployed** tables **independently of one another.**

Contract

Monolithic EagleEye

tables

application

Organization tables

Single EagleEye database

Assets

License service

service

License tables

Contract service

Organization service

**Each service owns all the data within their domain. This does not mean that each service has their own database. It just means** Assets **that only services that own** tables **that domain can access the database tables within it.**

**Licensed to <null>**

Contract tables

Organization tables

**42** CHAPTER 2 ***Building microservices with Spring Boot***

After you’ve broken a problem domain down into discrete pieces, you’ll often find yourself struggling to determine whether you’ve achieved the right level of granularity for your services. A microservice that’s too coarse- or fine-grained will have a number of telltale attributes that we’ll discuss shortly.

When you’re building a microservice architecture, the question of granularity is important, but you can use the following concepts to determine the correct solution:

1 *It’s better to start broad with your microservice and refactor to smaller services—*It’s easy to go overboard when you begin your microservice journey and make every- thing a microservice. But decomposing the problem domain into small services often leads to premature complexity because microservices devolve into noth- ing more than 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’s easier to refactor from being too coarse-grained to being too fine-grained. 3 *Service responsibilities will change over time as your understanding of the problem domain grows—*Often, a microservice gains responsibilities as new application functionality is requested. What starts as a single microservice might grow into multiple services, with the original microservice acting as an orchestration layer for these new services and encapsulating their functionality from other parts of the application.

The smells of a bad microservice How do you know whether your microservices are the right size? If a microservice is too coarse-grained, you’ll likely see the following: *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 array of business rules. *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 mul- tiple tables or reaching out to tables outside of the immediate database, this is a clue the service is too big. I like to use the guideline that a microservice should own no more than three to five tables. Any more, and your service is likely to have too much responsibility. *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 hun- dreds of unit and integration test cases, you might need to refactor. What about a microservice that’s too fine-grained? *The 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 because the number of services needed to get a piece of work done grows tremendously. A common smell is when you have dozens of microser- vices in an application and each service interacts with only a single database table.

**Licensed to <null>**

**43 *The architect’s story: designing the microservice architecture***

*Your microservices are heavily interdependent on one another—*You find that the microservices in one part of the problem domain keep calling back and forth between each other to complete a single user request.

*Your microservices become a collection of simple CRUD (Create, Replace, Update, Delete) services*—Microservices are an expression of business logic and not an abstraction layer over your data sources. If your microservices do nothing but CRUD- related logic, they’re probably too fine-grained.

A microservices architecture should be developed with an evolutionary thought pro- cess where you know that you aren’t going to get the design right the first time. That’s why it’s better to start with your first set of services being more coarse-grained than fine-grained. It’s also important not to be dogmatic with your design. You may run into physical constraints on your services where you’ll need to make an aggregation service that joins data together because two separate services will be too chatty, or where no clear boundaries exist between the domain lines of a service.

In the end, take a pragmatic approach and deliver, rather than waste time trying to get the design perfect and then 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 learning one or two of the services in the application.

In general, the following guidelines can be used for thinking about service inter- face 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 standard HTTP verbs (GET, PUT, POST, and DELETE). Model your basic behav- iors around these HTTP verbs. 2 *Use URI’s to communicate intent—*The URI you use as endpoints for the service should describe the different resources in your problem domain and provide a basic mechanism for relationships of resources within your problem domain. 3 *Use JSON for your requests and responses—*JavaScript Object Notation (in other words, JSON) is an extremely lightweight data-serialization protocol and is 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 your services.

**Licensed to <null>**

**44** CHAPTER 2 ***Building microservices with Spring Boot***

All the basic guidelines drive to one thing, making your service interfaces easy to understand and consumable. You want a developer to sit down and look at the service interfaces and start using them. If a microservice isn’t 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’ve spent this chapter talking about why microservices are a powerful architectural pattern for building applications. But I haven’t touched on when you shouldn’t use microservices to build your applications. Let’s walk through 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), they introduce a level of complexity into your application that wouldn’t be there in more monolithic appli- cations. Microservice architectures require a high degree of operational maturity. Don’t consider using microservices unless your organization is willing to invest in the automation and operational work (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. In a large microservices-based applica- tion, you might end up with 50 to 100 servers or containers (usually virtual) that have to be built and maintained in production alone. Even with the lower cost of running these services in the cloud, the operational complexity of having to manage and mon- itor these servers can be tremendous.

NOTE 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 toward reusability and are extremely useful for building large applications that need to be highly resilient and scalable. This is one of the rea- sons why so many cloud-based companies have adopted microservices. If you’re build- ing small, departmental-level applications or applications with a small user base, the complexity associated with building on a distributed model such as microservices might be more expense then it’s worth.

**Licensed to <null>**

**45 *The developer’s tale: building a microservice with Spring Boot and Java***

***2.2.4 Data transformations and consistency***

As you begin looking at microservices, you need to think through the data usage pat- terns of your services and how service consumers are going to use them. A microser- vice wraps around and abstracts away a small number of tables and works well as a mechanism for performing “operational” tasks such as creating, adding, and perform- ing 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 on too much responsibility and can also become vulnerable to performance problems.

Also keep in mind that no standard exists for performing transactions across microservices. If you need transaction management, you will need to build that logic yourself. In addition, as you’ll see in chapter 7, microservices can communicate amongst themselves by using messages. Messaging introduces latency in data updates. Your applications need 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 implementa- tion space requires a shift in perspective. Specifically, as a developer, you need to establish a basic pattern of how each of the microservices in your application is going to be implemented. While each service is going to be unique, you want to make sure that you’re using a framework that removes boilerplate code and that each piece of your microservice is laid out in the same consistent fashion.

In this section, we’ll explore the developer’s priorities in building the licensing microservice from your EagleEye domain model. Your licensing service is going to be written using Spring Boot. Spring Boot is an abstraction layer over the standard Spring libraries that allows developers to quickly build Groovy- and Java-based web applications and microservices with significantly less ceremony and configuration than a full-blown Spring application.

For your licensing service example, you’ll use Java as your core programming lan- guage and Apache Maven as your build tool.

Over the next several sections you’re 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

**Licensed to <null>**

**46** CHAPTER 2 ***Building microservices with Spring Boot***

***2.3.1 Getting started with the skeleton project***

To begin, you’ll create a skeleton project for the licensing. You can either pull down the source code down from GitHub (https://github.com/carnellj/spmia-chapter2) or create a licensing-service project directory with the following directory structure:

▪ licensing-service

▪ src/main/java/com/thoughtmechanix/licenses

▪ controllers

▪ model

▪ services

▪ resources

Once you’ve pulled down or created this directory structure, begin by writing your Maven script for the project. This will be the pom.xml file located at the root of the project directory. The following listing shows the Maven POM file for your 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 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>EagleEye Licensing Service</name> <description>Licensing Service</description> **Tells Maven to include the Spring Boot Starter** <parent> **Kit dependencies** <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-parent</artifactId>

<version>1.4.4.RELEASE</version> <relativePath/> </parent> **Tells Maven to include** <dependencies> **the Spring Boot** <dependency> **web dependencies** <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-web</artifactId> **Tells Maven to** </dependency>

**include the** <dependency> **Spring Actuator** <groupId>org.springframework.boot</groupId>

**dependencies**

<artifactId>spring-boot-starter-actuator</artifactId> </dependency> </dependencies>

**Licensed to <null>**

**47 *The developer’s tale: building a microservice with Spring Boot and Java***

<!—-Note: Some the build properties and Docker build plugins have been

excluded from the pom.xml in this pom (not in the source code in the github repository) because they are not relevant to our discussion here. -->

<build>

<plugins> <plugin>

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

We won’t go through the entire script in detail, but note a few key areas as we begin. Spring Boot is broken into many individual projects. The philosophy is that you shouldn’t have to “pull down the world” if you aren’t going to 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 part 1 of the Maven POM you tell Maven that you need to pull down version 1.4.4 of the Spring Boot framework

In parts 2 and 3 of the Maven file, you identify that you’re 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’ll find that as you build more functionality into your services, the list of these dependent projects becomes longer.

Also, Spring Source has provided Maven plugins that simplify the build and deploy- ment of the Spring Boot applications. Step 4 tells your 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 your interaction between Maven and Spring Boot. Finally, you’ll see a comment that sections of the Maven file have been removed. For the sake of the trees, I didn’t include the Spotify Docker plugins in listing 2.1.

NOTE Every chapter in this book includes Docker files for building and deploying the application as Docker containers. You can find details of how to build these Docker images in the README.md file in the code sections of each chapter.

***2.3.2 Booting your Spring Boot application: writing the Bootstrap class***

Your goal is to get a simple microservice up and running in Spring Boot and then iter- ate on it to deliver functionality. To this end, you need to create two classes in your licensing service microservice:

▪ A Spring Bootstrap class that will be used by Spring Boot to start up and initial- ize the application

▪ A Spring Controller class that will expose the HTTP endpoints that can be invoked on the microservice

**Tells Maven to include Spring specific maven plugins for building and deploying Spring Boot applications**

**Licensed to <null>**

**48** CHAPTER 2 ***Building microservices with Spring Boot***

As you’ll see shortly, Spring Boot uses annotations to simplify setting up and configur- ing the service. This becomes evident as you look at the bootstrap class in the follow- ing listing. This bootstrap class is 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** public class Application {

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

}

**@SpringBootApplication tells the Spring Boot framework that this is the bootstrap class for the project**

}**Call to start the entire Spring Boot service**

The first thing to note in this code is the use of the @SpringBootApplication annotation. Spring Boot uses this annotation 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 anno-

tation tag 2 Annotating a class with a @Configuration tag and then defining a constructor

method for each Spring Bean you want to build with a @Bean tag.

Under the covers, the @SpringBootApplication annotation marks the Application class in listing 2.2 as a configuration class, then begins auto-scanning all the classes on the Java class path for other Spring Beans.

The second thing to note is the Application class’s main() method. In the main() method, the SpringApplication.run(Application.class, args), the call starts the Spring container and returns a Spring ApplicationContext object. (You aren’t doing anything with the ApplicationContext, so it isn’t shown in the code.) The easiest thing to remember about the @SpringBootApplication annotation and the corresponding Application class is that it’s 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 you’ve gotten the build script out of the way and implemented a simple Spring Boot Bootstrap class, you can begin writing your first code that will do some- thing. This code will be your Controller class. In a Spring boot application, a

**Licensed to <null>**

**49 *The developer’s tale: building a microservice with Spring Boot and Java***

Controller class exposes the services endpoints and maps the data from an incom- ing HTTP request to a Java method that will process the request.

Give it a REST All the microservices in this book follow the REST approach to building your services. An in-depth discussion of REST is outside of the scope this book,a but for your pur- poses, all the services you build will have the following characteristics:

*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.

*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 CRUD functions found in most services.

*Use JSON as the serialization format for all data going to and from the service*—This isn’t a hard-and-fast principle for REST-based microservices, but JSON has become lingua franca for serializing data that’s going to be submitted and returned by a microservice. XML can be used, but many REST-based applications make heavy use of JavaScript and JSON (JavaScript Object Notation). JSON is the native format for serializing and deserializing data being consumed by JavaScript-based web front-ends and services.

*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 ser- vice. REST-based services take advantage of these HTTP status codes and other web- based infrastructure, such as reverse proxies and caches, which can be integrated with your microservices with relative ease.

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.

a Probably the most comprehensive coverage of the design of REST services is the book *REST in Practice* by Ian Robinson, et al (O’Reilly, 2010).

Your first controller class is 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 a set of annotations that keeps the effort needed to expose your service endpoints to a min- imum and allows you to focus on building the business logic for the service. We’ll start by looking at the basic controller class definition without any class methods in it yet. The following listing shows the controller class that you built for your licensing service.

Listing 2.3 Marking the **LicenseServiceController** as a Spring **RestController**

package com.thoughtmechanix.licenses.controllers;

import ... // Removed for conciseness

**Licensed to <null>**

**50** CHAPTER 2 ***Building microservices with Spring Boot***

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

//Body of the class removed for conciseness }**@RestController tells Spring Boot this is a REST-based services and will automatically serialize/deserialize service request/response to JSON.**

We’ll begin our exploration by looking at the @RestController annotation. The @RestController is a class-level Java annotation and tells the Spring Container that this Java class is going to be used for a REST-based service. This annotation automati- cally handles the serialization of data passed 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 doesn’t require you as the developer to return a ResponseBody class from your con- troller class. This is all handled by the presence of the @RestController annotation, which includes the @ResponseBody annotation.

Why JSON for microservices? Multiple protocols can be used to send data back and forth between HTTP-based microservices. JSON has emerged as the de facto standard for several reasons.

First, compared to other protocols such as the XML-based SOAP (Simple Object Access Protocol), it’s extremely lightweight in that you can express your data without having much textual overhead.

Second, it’s easily read and consumed by a human being. This is an underrated qual- ity for choosing a serialization protocol. When a problem arises, it’s critical for devel- opers to look at a chunk of JSON and quickly, visually process what’s in it. The simplicity of the protocol makes this incredibly easy to do.

Third, JSON is the default serialization protocol used in JavaScript. Since the dra- matic rise of JavaScript as a programming language and the equally dramatic rise of Single Page Internet Applications (SPIA) that rely heavily on JavaScript, JSON has become a natural fit for building REST-based applications because it’s what the front- end web clients use to call services.

Other mechanisms and protocols are more efficient than JSON for communicating between services. The Apache Thrift (http://thrift.apache.org) framework allows you to build multi-language services that can communicate with one another using a binary protocol. The Apache Avro protocol (http://avro.apache.org) is a data serializa- tion protocol that converts data back and forth to a binary format between client and server calls.

If you need to minimize the size of the data you’re sending across the wire, I recom- mend you look at these protocols. But it has been my experience that using straight- up JSON in your microservices works effectively and doesn’t interpose another layer of communication to debug between your service consumers and service clients.

**Exposes all the HTTP endpoints in this class with a prefix of /v1/organizations/(organizationId}/licenses**

**Licensed to <null>**

**51 *The developer’s tale: building a microservice with Spring Boot and Java***

The second annotation shown in listing 2.3 is the @RequestMapping annotation. You can use the @RequestMapping annotation as a class-level and method-level annota- tion. 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’re establishing the root of the URL for all the other endpoints exposed by the controller.

In listing 2.3, the @RequestMapping(value="/v1/organizations/{organi- zationId}/licenses") uses the value attribute to establish the root of the URL for all endpoints exposed in the controller class. 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 how you expect the URL to be parameterized with an organizationId passed in every call. The use of organizationId in the URL allows you to differentiate between the different customers who might use your service.

Now you’ll add the first method to your controller. This method will implement the GET verb used in a REST call and return a single License class instance, as shown in the following listing. (For purposes of this discussion you’ll instantiate a Java class called License.)

Listing 2.4 Exposing an individual GET HTTP endpoint**Creates a GET endpoint with the value v1/organizations/{organizationId}/licenses{licenseId}**

**@RequestMapping(value="/{licenseId}",method = RequestMethod.GET)** public License getLicenses**(**

**@PathVariable("organizationId")** String organizationId, **@PathVariable("licenseId")** String licenseId) {

return new License() .withId(licenseId) .withProductName("Teleco") .withLicenseType("Seat") .withOrganizationId("TestOrg");

**Maps two parameters from the URL (organizationId and licenseId) to method parameters**

}The first thing you’ve done in this listing 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, you’re building on the root-level annotation specified at the top of the class to match all HTTP requests coming to the controller with the endpoint /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 previous example, you’re matching on the GET method as represented by the RequestMethod.GET enumeration.

The second thing to note about listing 2.4 is that you use the @PathVariable anno- tation in the parameter body of the getLicenses() method. (2) The @Path- Variable annotation is used to map the parameter values passed in the incoming URL

**Licensed to <null>**

**52** CHAPTER 2 ***Building microservices with Spring Boot***

(as denoted by the {parameterName} syntax) to the parameters of your method. In your code example from listing 2.4, you’re 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 standards for the end- points that will 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 exist between the resources managed within the service. I’ve found the following guidelines useful for 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 intui- tive and easier to use. Be consistent in your naming conventions. 2 *Use the URL to establish relationships between resources—*Oftentimes you’ll have a parent-child relationship between resources within your microservices where the child doesn’t exist outside the context of the parent (hence you might not have a separate microservice for the child). Use the URLs to express these relationships. But if you find that your URLs tend to be exces- sively long and nested, your microservice may be 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 prepend 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 consum- ers using them.

At this point you have something you can call as a service. From a command line win- dow, go to your project directory where you’ve downloaded the sample code and exe- cute 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

**The license server starting on port 8080**

**Licensed to <null>**

**53 *The DevOps story: building for the rigors of runtime***

Once the service is started, you can directly hit the exposed endpoint. Because your first method exposed is a GET call, you can use a number of methods for invoking the service. My preferred method is to use a chrome-based tool like POSTMAN or CURL for calling the service. Figure 2.5 shows a GET performed on the http://local- host:8080/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/ licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a endpoint.

Figure 2.5 Your licensing service being called with POSTMAN

At this point you have a running skeleton of a service. But from a development per- spective, this service isn’t complete. A good microservice design doesn’t eschew segre- gating the service into well-defined business logic and data access layers. As you progress in later chapters, you’ll continue to iterate on this service and delve further into how to structure it.

Let’s switch to the final perspective: exploring how a DevOps engineer would oper- ationalize the service and package it for deployment to the cloud.

***2.4 The DevOps story: building for the rigors of runtime***

For the DevOps engineer, the design of the microservice is all about managing the ser- vice 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, you’ll start your microservice devel- opment effort with four principles and build on these principles later in the book. These principles are

1 A microservice should be *self-contained* and *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 read the data it needs to configure itself from a central location or have

**When the GET endpoint is called, a JSON payload containing licensing data is returned.**

**Licensed to <null>**

**54** CHAPTER 2 ***Building microservices with Spring Boot***

its configuration information passed on 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 *communicate* its health. This is a critical part of your cloud architecture. Microservice instances will fail and clients need to route around bad service instances.

These four principles expose the paradox that can exist with microservice develop- ment. Microservices are smaller in size and scope, but their use introduces more mov- ing parts in an application, especially because microservices are distributed and running independently of each other in their own distributed containers. This intro- duces a high degree of coordination and more opportunities for failure points in the application.

From a DevOps perspective, you must address the operational needs of a microser- vice up front and translate these four principles into a standard set of lifecycle events that occur every time a microservice is built and deployed to an environment. The four principles can be mapped to the following operational lifecycle steps:

▪ *Service assembly*—How do you package and deploy your service to guarantee repeatability and consistency so that the same service code and runtime is deployed exactly the same way?

▪ *Service bootstrapping*—How do you separate your application and environment- specific configuration code from the runtime code so you can start and deploy a microservice instance quickly in any environment without human interven- tion to configure the microservice?

▪ *Service registration/discovery*—When a new microservice instance is deployed, how do you make the new service instance discoverable by other application clients?

▪ *Service monitoring*—In a microservices environment it’s extremely common for multiple instances of the same service to be running due to high availability needs. From a DevOps perspective, you need to monitor microservice instances and ensure that any faults in your microservice are routed around and that ail- ing service instances are taken down.

Figure 2.6 shows how these four steps fit together.

**Licensed to <null>**

**55 *The DevOps story: building for the rigors of runtime***

1. Assembly

2. Bootstrapping

3. Discovery 4. Monitoring

Build/deploy

Executable

Configuration

Service discovery

Service discovery engine

JAR

repository

agent

agent

Failing

Source code

Service instance startup

Multiple service

Multiple service repository

instances

instances

Service client

Figure 2.6 When a microservice starts up, it goes through multiple steps in its lifecycle.

Building the Twelve-Factor microservice service application One of my biggest hopes with this book is that you realize that a successful microser- vice architecture requires strong application development and DevOps practices. One of the most succinct summaries of these practices can be found in Heroku’s Twelve- Factor Application manifesto (https://12factor.net/). This document provides 12 best practices you should always keep in the back of your mind when building microser- vices. As you read this book, you’ll see these practices intertwined into the examples. I’ve summarized them as follows:

*Codebase*—All application code and server provisioning information should be in ver- sion control. Each microservice should have its own independent code repository within the source control systems.

*Dependencies*—Explicitly declare the dependencies your application uses through build tools such as Maven (Java). Third-party JAR dependence should be declared using their specific version numbers. This allows your microservice to always be built using the same version of libraries.

*Config*—Store your application configuration (especially your environment-specific configuration) independently from your code. Your application configuration should never be in the same repository as your source code.

*Backing services*—Your microservice will often communicate over a network to a data- base or messaging system. When it does, you should ensure that at any time, you can swap out your implementation of the database from an in-house managed service to a third-party service. In chapter 10, we demonstrate this when you move your ser- vices away from a locally managed Postgres database to one managed by Amazon.

**Licensed to <null>**