

Covers Spring 5.0

Spring

IN ACTION

FIFTH EDITION

Craig Walls



MANNING

Praise for Spring in Action, 4th edition

“The best book for Spring—updated and revised.”

—Gregor Zurowski, Sotheby’s

“The classic, remastered and full of awesomeness.”

—Mario Arias, Cake Solutions Ltd.

“Informative, accurate, and insightful!”

—Jeelani Shaik, [D3Banking.com](#)

“After ten years, this is still the clearest and most comprehensive introduction to the core concepts of the Spring platform.”

—James Wright, Sword-Apak

“This book is a quick and easy way to get into the Spring Framework Universe. Simply perfect for Java developers.”

—Jens O’Richter, freelance Senior Software Architect

“This book belongs on the bookshelf of any serious Java developer who uses Spring.”

—Jonathan Thoms, Expedia Inc.

“Spring in Action is an excellent travel companion for the huge landscape that is the Spring Framework.”

—Ricardo Lima, Senado Federal do Brasil

“Pragmatic advice for Java’s most important framework.”

—Mike Roberts, Information Innovators

Spring in Action

Fifth Edition

COVERS SPRING 5.0

CRAIG WALLS



MANNING
SHELTER ISLAND

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

©2019 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.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Jennifer Stout
Project manager: Janet Vail
Copy editors: Frances Buran, Andy Carroll
Proofreaders: Melody Dolab, Katie Tennant
Technical proofreader: Joshua White
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617294945

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – DP – 23 22 21 20 19 18

brief contents

PART 1 FOUNDATIONAL SPRING	1
1 ■ Getting started with Spring	3
2 ■ Developing web applications	29
3 ■ Working with data	56
4 ■ Securing Spring	84
5 ■ Working with configuration properties	114
PART 2 INTEGRATED SPRING	135
6 ■ Creating REST services	137
7 ■ Consuming REST services	169
8 ■ Sending messages asynchronously	178
9 ■ Integrating Spring	209
PART 3 REACTIVE SPRING	239
10 ■ Introducing Reactor	241
11 ■ Developing reactive APIs	269
12 ■ Persisting data reactively	296

PART 4 CLOUD-NATIVE SPRING	321
13 ■ Discovering services	323
14 ■ Managing configuration	343
15 ■ Handling failure and latency	376
PART 5 DEPLOYED SPRING	393
16 ■ Working with Spring Boot Actuator	395
17 ■ Administering Spring	429
18 ■ Monitoring Spring with JMX	446
19 ■ Deploying Spring	454

contents

preface *xiii*
acknowledgments *xv*
about this book *xvii*

PART 1 FOUNDATIONAL SPRING 1

1	<i>Getting started with Spring</i>	3
1.1	What is Spring?	4
1.2	Initializing a Spring application	6
	<i>Initializing a Spring project with Spring Tool Suite</i>	7
	<i>Examining the Spring project structure</i>	11
1.3	Writing a Spring application	17
	<i>Handling web requests</i>	17
	<i>Defining the view</i>	19
	<i>Testing the controller</i>	20
	<i>Building and running the application</i>	21
	<i>Getting to know Spring Boot DevTools</i>	23
	<i>Let's review</i>	25
1.4	Surveying the Spring landscape	26
	<i>The core Spring Framework</i>	26
	<i>Spring Boot</i>	26
	<i>Spring Data</i>	27
	<i>Spring Security</i>	27
	<i>Spring Integration and Spring Batch</i>	27
	<i>Spring Cloud</i>	28

2 Developing web applications 29

- 2.1 Displaying information 30
 - Establishing the domain* 31 ▪ *Creating a controller class* 32
 - Designing the view* 35
- 2.2 Processing form submission 40
- 2.3 Validating form input 45
 - Declaring validation rules* 46 ▪ *Performing validation at form binding* 48 ▪ *Displaying validation errors* 49
- 2.4 Working with view controllers 51
- 2.5 Choosing a view template library 52
 - Caching templates* 54

3 Working with data 56

- 3.1 Reading and writing data with JDBC 57
 - Adapting the domain for persistence* 59 ▪ *Working with JdbcTemplate* 60 ▪ *Defining a schema and preloading data* 64
 - Inserting data* 66
- 3.2 Persisting data with Spring Data JPA 75
 - Adding Spring Data JPA to the project* 76 ▪ *Annotating the domain as entities* 76 ▪ *Declaring JPA repositories* 80
 - Customizing JPA repositories* 81

4 Securing Spring 84

- 4.1 Enabling Spring Security 85
- 4.2 Configuring Spring Security 86
 - In-memory user store* 88 ▪ *JDBC-based user store* 89
 - LDAP-backed user store* 92 ▪ *Customizing user authentication* 96
- 4.3 Securing web requests 103
 - Securing requests* 104 ▪ *Creating a custom login page* 106
 - Logging out* 109 ▪ *Preventing cross-site request forgery* 109
- 4.4 Knowing your user 110

5 Working with configuration properties 114

- 5.1 Fine-tuning autoconfiguration 115
 - Understanding Spring's environment abstraction* 116
 - Configuring a data source* 117 ▪ *Configuring the embedded server* 119 ▪ *Configuring logging* 120 ▪ *Using special property values* 121

5.2	Creating your own configuration properties	122
	<i>Defining configuration properties holders</i>	124
	<i>Declaring configuration property metadata</i>	126
5.3	Configuring with profiles	129
	<i>Defining profile-specific properties</i>	130
	<i>Activating profiles</i>	131
	<i>Conditionally creating beans with profiles</i>	132

PART 2 INTEGRATED SPRING 135

6 Creating REST services 137

6.1	Writing RESTful controllers	138
	<i>Retrieving data from the server</i>	140
	<i>Sending data to the server</i>	145
	<i>Updating data on the server</i>	146
	<i>Deleting data from the server</i>	148
6.2	Enabling hypermedia	149
	<i>Adding hyperlinks</i>	152
	<i>Creating resource assemblers</i>	154
	<i>Naming embedded relationships</i>	159
6.3	Enabling data-backed services	160
	<i>Adjusting resource paths and relation names</i>	162
	<i>Paging and sorting</i>	164
	<i>Adding custom endpoints</i>	165
	<i>Adding custom hyperlinks to Spring Data endpoints</i>	167

7 Consuming REST services 169

7.1	Consuming REST endpoints with RestTemplate	170
	<i>GETting resources</i>	172
	<i>PUTting resources</i>	173
	<i>DELETEing resources</i>	174
	<i>POSTing resource data</i>	174
7.2	Navigating REST APIs with Traverson	175

8 Sending messages asynchronously 178

8.1	Sending messages with JMS	179
	<i>Setting up JMS</i>	179
	<i>Sending messages with JmsTemplate</i>	181
	<i>Receiving JMS messages</i>	188
8.2	Working with RabbitMQ and AMQP	192
	<i>Adding RabbitMQ to Spring</i>	193
	<i>Sending messages with RabbitTemplate</i>	194
	<i>Receiving message from RabbitMQ</i>	198
8.3	Messaging with Kafka	202
	<i>Setting up Spring for Kafka messaging</i>	203
	<i>Sending messages with KafkaTemplate</i>	204
	<i>Writing Kafka listeners</i>	206

9 *Integrating Spring* 209

- 9.1 Declaring a simple integration flow 210
 - Defining integration flows with XML* 211 ▪ *Configuring integration flows in Java* 213 ▪ *Using Spring Integration's DSL configuration* 215
- 9.2 Surveying the Spring Integration landscape 216
 - Message channels* 217 ▪ *Filters* 219 ▪ *Transformers* 220
 - Routers* 221 ▪ *Splitters* 223 ▪ *Service activators* 225
 - Gateways* 227 ▪ *Channel adapters* 228 ▪ *Endpoint modules* 230
- 9.3 Creating an email integration flow 231

PART 3 REACTIVE SPRING 239

10 *Introducing Reactor* 241

- 10.1 Understanding reactive programming 242
 - Defining Reactive Streams* 243
- 10.2 Getting started with Reactor 245
 - Diagramming reactive flows* 246 ▪ *Adding Reactor dependencies* 247
- 10.3 Applying common reactive operations 248
 - Creating reactive types* 249 ▪ *Combining reactive types* 253
 - Transforming and filtering reactive streams* 257 ▪ *Performing logic operations on reactive types* 266

11 *Developing reactive APIs* 269

- 11.1 Working with Spring WebFlux 269
 - Introducing Spring WebFlux* 271 ▪ *Writing reactive controllers* 272
- 11.2 Defining functional request handlers 276
- 11.3 Testing reactive controllers 279
 - Testing GET requests* 279 ▪ *Testing POST requests* 282
 - Testing with a live server* 284
- 11.4 Consuming REST APIs reactively 285
 - GETting resources* 285 ▪ *Sending resources* 287
 - Deleting resources* 288 ▪ *Handling errors* 289
 - Exchanging requests* 290

11.5 Securing reactive web APIs 292

Configuring reactive web security 292 ▪ *Configuring a reactive user details service* 294

12 Persisting data reactively 296

12.1 Understanding Spring Data's reactive story 297

Spring Data reactive distilled 297 ▪ *Converting between reactive and non-reactive types* 298 ▪ *Developing reactive repositories* 300

12.2 Working with reactive Cassandra repositories 300

Enabling Spring Data Cassandra 301 ▪ *Understanding Cassandra data modeling* 303 ▪ *Mapping domain types for Cassandra persistence* 304 ▪ *Writing reactive Cassandra repositories* 309

12.3 Writing reactive MongoDB repositories 312

Enabling Spring Data MongoDB 312 ▪ *Mapping domain types to documents* 314 ▪ *Writing reactive MongoDB repository interfaces* 317

PART 4 CLOUD-NATIVE SPRING.....321

13 Discovering services 323

13.1 Thinking in microservices 324

13.2 Setting up a service registry 326

Configuring Eureka 330 ▪ *Scaling Eureka* 333

13.3 Registering and discovering services 334

Configuring Eureka client properties 335 ▪ *Consuming services* 337

14 Managing configuration 343

14.1 Sharing configuration 344

14.2 Running Config Server 345

Enabling Config Server 346 ▪ *Populating the configuration repository* 349

14.3 Consuming shared configuration 352

14.4 Serving application- and profile-specific properties 353

Serving application-specific properties 354 ▪ *Serving properties from profiles* 355

14.5 Keeping configuration properties secret 357

Encrypting properties in Git 357 ▪ *Storing secrets in Vault* 360

14.6 Refreshing configuration properties on the fly	364
<i>Manually refreshing configuration properties</i>	365
<i>Automatically refreshing configuration properties</i>	367

15 *Handling failure and latency* 376

15.1 Understanding circuit breakers	376	
15.2 Declaring circuit breakers	378	
<i>Mitigating latency</i>	381 ▪ <i>Managing circuit breaker thresholds</i>	382
15.3 Monitoring failures	383	
<i>Introducing the Hystrix dashboard</i>	384 ▪ <i>Understanding Hystrix thread pools</i>	387
15.4 Aggregating multiple Hystrix streams	389	

PART 5 DEPLOYED SPRING 393

16 *Working with Spring Boot Actuator* 395

16.1 Introducing Actuator	396		
<i>Configuring Actuator's base path</i>	397 ▪ <i>Enabling and disabling Actuator endpoints</i>	398	
16.2 Consuming Actuator endpoints	399		
<i>Fetching essential application information</i>	400 ▪ <i>Viewing configuration details</i>	403 ▪ <i>Viewing application activity</i>	411
<i>Tapping runtime metrics</i>	413		
16.3 Customizing Actuator	416		
<i>Contributing information to the /info endpoint</i>	416		
<i>Defining custom health indicators</i>	421 ▪ <i>Registering custom metrics</i>	422 ▪ <i>Creating custom endpoints</i>	424
16.4 Securing Actuator	426		

17 *Administering Spring* 429

17.1 Using the Spring Boot Admin	430		
<i>Creating an Admin server</i>	430 ▪ <i>Registering Admin clients</i>	431	
17.2 Exploring the Admin server	435		
<i>Viewing general application health and information</i>	436		
<i>Watching key metrics</i>	437 ▪ <i>Examining environment properties</i>	438 ▪ <i>Viewing and setting logging levels</i>	439
<i>Monitoring threads</i>	440 ▪ <i>Tracing HTTP requests</i>	441	

17.3 Securing the Admin server 442

Enabling login in the Admin server 443 ▪ *Authenticating with the Actuator* 444

18 *Monitoring Spring with JMX* 446

- 18.1 Working with Actuator MBeans 446
- 18.2 Creating your own MBeans 449
- 18.3 Sending notifications 451

19 *Deploying Spring* 454

- 19.1 Weighing deployment options 455
- 19.2 Building and deploying WAR files 456
- 19.3 Pushing JAR files to Cloud Foundry 458
- 19.4 Running Spring Boot in a Docker container 461
- 19.5 The end is where we begin 465

appendix Bootstrapping Spring applications 466

index 487

preface

After nearly 15 years of working with Spring and having written five editions of this book (not to mention *Spring Boot in Action*), you'd think that it'd be hard to come up with something exciting and new to say about Spring when writing the preface for this book. But nothing could be further from the truth!

Every single release of Spring, Spring Boot, and all of the other projects in the Spring ecosystem unleashes some new amazing capabilities that rekindle the fun in developing applications. With Spring reaching a significant milestone with its 5.0 release and Spring Boot releasing version 2.0, there's so much more Spring to enjoy that it was a no-brainer to write another edition of *Spring in Action*.

The big story of Spring 5 is reactive programming support, including Spring WebFlux, a brand new reactive web framework that borrows its programming model from Spring MVC, allowing developers to create web applications that scale better and make better use of fewer threads. Moving toward the backend of a Spring application, the latest edition of Spring Data enables the creation of reactive, non-blocking data repositories. And all of this is built on top of Project Reactor, a Java library for working with reactive types.

In addition to the new reactive programming features of Spring 5, Spring Boot 2 now provides even more autoconfiguration support than ever before as well as a completely reimaged Actuator for peeking into and manipulating a running application.

What's more, as developers look to break down their monolithic applications into discrete microservices, Spring Cloud provides facilities that make it easy to configure and discover microservices, as well as fortify them so they're more resilient to failure.

I'm happy to say that this fifth edition of Spring in Action covers all of this and more! If you're a seasoned veteran with Spring, *Spring in Action, Fifth Edition* will be your guide to everything new that Spring has to offer. On the other hand, if you're new to Spring, then there's no better time than now to get in on the action and the first few chapters will get you up and running in no time!

It's been an exciting 15 years of working with Spring. And now that I've written this fifth edition of Spring in Action, I'm eager to share that excitement with you!

acknowledgments

One of the most amazing things that Spring and Spring Boot do is to automatically provide all of the foundational plumbing for an application, leaving you as a developer to focus primarily on the logic that's unique to your application. Unfortunately, no such magic exists for writing a book. Or does it?

At Manning, there were several people working their magic to make sure that this book is the best it can possibly be. Many thanks in particular to Jenny Stout, my development editor, and to the production team, including project manager Janet Vail, copyeditors Andy Carroll and Frances Buran, and proofreaders Katie Tennant and Melody Dolab. Thanks, too, to technical proofer Joshua White who was thorough and helpful.

Along the way, we got feedback from several peer reviewers who made sure that the book stayed on target and covered the right stuff. For this, my thanks goes to Andrea Barisone, Arnaldo Ayala, Bill Fly, Colin Joyce, Daniel Vaughan, David Witherspoon, Eddu Melendez, Iain Campbell, Jetro Coenradie, John Gunvaldson, Markus Matzker, Nick Rakochy, Nusry Firdousi, Piotr Kafel, Raphael Villela, Riccardo Noviello, Sergio Fernandez Gonzalez, Sergiy Pylypets, Thiago Presa, Thorsten Weber, Waldemar Modzelewski, Yagiz Erkan, and Željko Trogrlić.

As always, there'd be absolutely no point in writing this book if it weren't for the amazing work done by the members of the Spring engineering team. I'm amazed at what you've created and how we continue to change how software is developed.

Many thanks to my fellow speakers on the No Fluff/Just Stuff tour. I continue to learn so much from every one of you. I especially want to thank Brian Sletten, Nate

Schutta, and Ken Kousen for conversations and emails about Spring that have helped shape this book.

Once again, I'd like to thank the Phoenicians. You know what you did.

Finally, to my beautiful wife Raymie, the love of my life, my sweetest dream, and my inspiration: Thank you for your encouragement and for putting up with another book project. And to my sweet and wonderful girls, Maisy and Madi: I am so proud of you and of the amazing young ladies you are becoming. I love all of you more than you can imagine or I can possibly express.

about this book

Spring in Action, Fifth Edition was written to equip you to build amazing applications using the Spring Framework, Spring Boot, and a variety of ancillary members of the Spring ecosystem. It begins by showing you how to develop web-based, database-backed Java applications with Spring and Spring Boot. It then expands on the essentials by showing how to integrate with other applications, program using reactive types, and then break an application into discrete microservices. Finally, it discusses how to ready an application for deployment.

Although all of the projects in the Spring ecosystem provide excellent documentation, this book does something that none of the reference documents do: provide a hands-on, project-driven guide to bringing the elements of Spring together to build a real application.

Who should read this book

Spring in Action, 5th edition is for Java developers who want to get started with Spring Boot and the Spring Framework as well as for seasoned Spring developers who want to go beyond the basics and learn the newest features of Spring.

How this book is organized: a roadmap

The book has 5 parts spanning 19 chapters. Part 1 covers the foundational topics of building Spring applications:

- Chapter 1 introduces Spring and Spring Boot and how to initialize a Spring project. In this chapter, you'll take the first steps toward building a Spring application that you'll expand upon throughout the course of the book.

- Chapter 2 discusses building the web layer of an application using Spring MVC. In this chapter, you'll build controllers that handle web requests and views that render information in the web browser.
- Chapter 3 delves into the backend of a Spring application where data is persisted to a relational database.
- In chapter 4, you'll use Spring Security to authenticate users and prevent unauthorized access to an application.
- Chapter 5 reveals how to configure a Spring application using Spring Boot configuration properties. You'll also learn how to selectively apply configuration using profiles.

Part 2 covers topics that help integrate your Spring application with other applications:

- Chapter 6 expands on the discussion of Spring MVC started in chapter 2 by looking at how to write REST APIs in Spring.
- Chapter 7 turns the tables on chapter 6 to show how a Spring application can consume a REST API.
- Chapter 8 looks at using asynchronous communication to enable a Spring application to both send and receive messages using the Java Message Service, RabbitMQ, or Kafka.
- Chapter 9 discusses declarative application integration using the Spring Integration project.

Part 3 explores the exciting new support for reactive programming in Spring:

- Chapter 10 introduces Project Reactor, the reactive programming library that underpins Spring 5's reactive features.
- Chapter 11 revisits REST API development, introducing Spring WebFlex, a new web framework that borrows much from Spring MVC while offering a new reactive model for web development.
- Chapter 12 takes a look at writing reactive data persistence with Spring Data to read and write data to Cassandra and Mongo databases.

Part 4 breaks down the monolithic application model, introducing you to Spring Cloud and microservice development:

- Chapter 13 dives into service discovery, using Spring with Netflix's Eureka registry to both register and discover Spring-based microservices.
- Chapter 14 shows how to centralize application configuration in a configuration server that shares configuration across multiple microservices.
- Chapter 15 introduces the circuit breaker pattern with Hystrix, enabling microservices that are resilient in the face of failure.

In part 5, you'll ready an application for production and see how to deploy it:

- Chapter 16 introduces the Spring Boot Actuator, an extension to Spring Boot that exposes the internals of a running Spring application as REST endpoints.

- In chapter 17 you'll see how to use the Spring Boot Admin to put a user-friendly browser-based administrative application on top of the Actuator.
- Chapter 18 discusses how to expose and consume Spring beans as JMX MBeans.
- Finally, in chapter 19 you'll see how to deploy your Spring application in a variety of production environments.

In general, developers new to Spring should start with chapter 1 and work through each chapter sequentially. Experienced Spring developers may prefer to jump in at any point that interests them. Even so, each chapter builds upon the previous chapter, so there may be some context missing if you dive into the middle of the book.

About the code

This book contains many examples of source code both in numbered listings and inline 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 was not 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.

Source code for the examples in this book is available for download from the publisher's website at www.manning.com/books/spring-in-action-fifth-edition as well as from the author's GitHub account at github.com/habuma/spring-in-action-5-samples.

Book forum

Purchase of *Spring in Action*, 5th edition, 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, go to <https://forums.manning.com/forums/spring-in-action-fifth-edition>. You can also learn more about Manning's forums and the rules of conduct at <https://forums.manning.com/forums/about>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue 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 contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

Need additional help?

- The Spring website has several useful getting-started guides (some of which were written by the author of this book) at <https://spring.io/guides>.
- The Spring tag at StackOverflow (<https://stackoverflow.com/questions/tagged/spring>) as well as the Spring Boot tag at StackOverflow are great places to ask questions and help others with Spring. Helping someone else with their Spring questions is a great way to learn Spring!

About the author

CRAIG WALLS is a principal engineer with Pivotal. He's a zealous promoter of the Spring Framework, speaking frequently at local user groups and conferences and writing about Spring. When he's not slinging code, Craig is planning his next trip to Disney World or Disneyland and spending as much time as he can with his wife, two daughters, two birds, and three dogs.

About the cover illustration

The figure on the cover of *Spring in Action*, 5th edition, is “Le Caraco,” or an inhabitant of the province of Karak in southwest Jordan. Its capital is the city of Al-Karak, which boasts an ancient hilltop castle with magnificent views of the Dead Sea and surrounding plains. The illustration is taken from a French travel book, *Encyclopédie des Voyages* by J. G. St. Sauveur, published in 1796. Travel for pleasure was a relatively new phenomenon at the time and travel guides such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other regions of France and abroad.

The diversity of the drawings in the *Encyclopédie des Voyages* speaks vividly of the distinctiveness and individuality of the world’s towns and provinces just two hundred years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other. The travel guide brings to life a sense of isolation and distance of that period, and of every other historic period except our own hyperkinetic present.

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 inhabitants of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life. We at Manning celebrate the inventiveness, the initiative, and the fun of the computer business with book covers based on the rich diversity of regional life two centuries ago brought back to life by the pictures from this travel guide.

Part 1

Foundational Spring

P

art 1 of this book will get you started writing a Spring application, learning the foundations of Spring along the way.

In chapter 1, I'll give you a quick overview of Spring and Spring Boot essentials and show you how to initialize a Spring project as you work on building Taco Cloud, your first Spring application. In chapter 2, you'll dig deeper into the Spring MCV and learn how to present model data in the browser and how to process and validate form input. You'll also get some tips on choosing a view template library. You'll add data persistence to the Taco Cloud application in chapter 3. There, we'll cover using Spring's JDBC template, how to insert data, and how to declare JPA repositories with Spring Data. Chapter 4 covers security for your Spring application, including autoconfiguring Spring Security, defining custom user storage, customizing the login page, and securing against cross-site request forgery (CSRF) attacks. To close out part 1, we'll look at configuration properties in chapter 5. You'll learn how to fine-tune autoconfigured beans, apply configuration properties to application components, and work with Spring profiles.

1

Getting started with Spring

This chapter covers

- Spring and Spring Boot essentials
- Initializing a Spring project
- An overview of the Spring landscape

Although the Greek philosopher Heraclitus wasn't well known as a software developer, he seemed to have a good handle on the subject. He has been quoted as saying, "The only constant is change." That statement captures a foundational truth of software development.

The way we develop applications today is different than it was a year ago, 5 years ago, 10 years ago, and certainly 15 years ago, when an initial form of the Spring Framework was introduced in Rod Johnson's book, *Expert One-on-One J2EE Design and Development* (Wrox, 2002, <http://mng.bz/oVjy>).

Back then, the most common types of applications developed were browser-based web applications, backed by relational databases. While that type of development is still relevant, and Spring is well equipped for those kinds of applications, we're now also interested in developing applications composed of microservices destined for the cloud that persist data in a variety of databases. And a new interest in reactive programming aims to provide greater scalability and improved performance with non-blocking operations.

As software development evolved, the Spring Framework also changed to address modern development concerns, including microservices and reactive programming. Spring also set out to simplify its own development model by introducing Spring Boot.

Whether you’re developing a simple database-backed web application or constructing a modern application built around microservices, Spring is the framework that will help you achieve your goals. This chapter is your first step in a journey through modern application development with Spring.

1.1 **What is Spring?**

I know you’re probably itching to start writing a Spring application, and I assure you that before this chapter ends, you’ll have developed a simple one. But first, let me set the stage with a few basic Spring concepts that will help you understand what makes Spring tick.

Any non-trivial application is composed of many components, each responsible for its own piece of the overall application functionality, coordinating with the other application elements to get the job done. When the application is run, those components somehow need to be created and introduced to each other.

At its core, Spring offers a *container*, often referred to as the *Spring application context*, that creates and manages application components. These components, or *beans*, are wired together inside the Spring application context to make a complete application, much like bricks, mortar, timber, nails, plumbing, and wiring are bound together to make a house.

The act of wiring beans together is based on a pattern known as *dependency injection* (DI). Rather than have components create and maintain the lifecycle of other beans that they depend on, a dependency-injected application relies on a separate entity (the container) to create and maintain all components and inject those into the beans that need them. This is done typically through constructor arguments or property accessor methods.

For example, suppose that among an application’s many components, there are two that you’ll address: an inventory service (for fetching inventory levels) and a product service (for providing basic product information). The product service depends on the inventory service to be able to provide a complete set of information about products. Figure 1.1 illustrates the relationships between these beans and the Spring application context.

On top of its core container, Spring and a full portfolio of related libraries offer a web framework, a variety of data persistence options, a security framework, integration with other systems, runtime monitoring, microservice support, a reactive programming model, and many other features necessary for modern application development.

Historically, the way you would guide Spring’s application context to wire beans together was with one or more XML files that described the components and their relationship to other components. For example, the following XML declares two

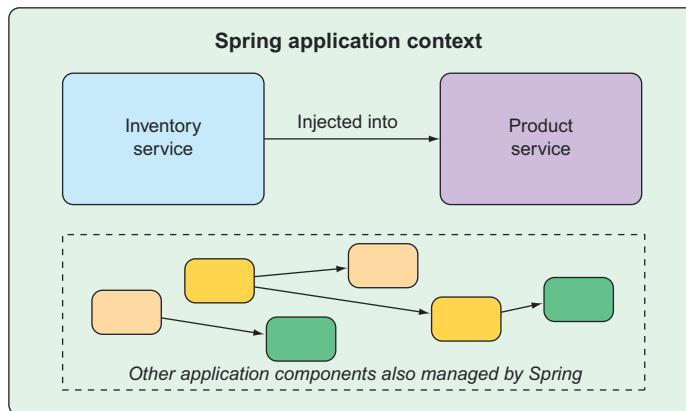


Figure 1.1 Application components are managed and injected into each other by the Spring application context.

beans, an `InventoryService` bean and a `ProductService` bean, and wires the `InventoryService` bean into `ProductService` via a constructor argument:

```
<bean id="inventoryService"
      class="com.example.InventoryService" />

<bean id="productService"
      class="com.example.ProductService" />
      <constructor-arg ref="inventoryService" />
</bean>
```

In recent versions of Spring, however, a Java-based configuration is more common. The following Java-based configuration class is equivalent to the XML configuration:

```
@Configuration
public class ServiceConfiguration {
    @Bean
    public InventoryService inventoryService() {
        return new InventoryService();
    }

    @Bean
    public ProductService productService() {
        return new ProductService(inventoryService());
    }
}
```

The `@Configuration` annotation indicates to Spring that this is a configuration class that will provide beans to the Spring application context. The configuration's class methods are annotated with `@Bean`, indicating that the objects they return should be added as beans in the application context (where, by default, their respective bean IDs will be the same as the names of the methods that define them).

Java-based configuration offers several benefits over XML-based configuration, including greater type safety and improved refactorability. Even so, explicit configuration with either Java or XML is only necessary if Spring is unable to automatically configure the components.

Automatic configuration has its roots in the Spring techniques known as *autowiring* and *component scanning*. With component scanning, Spring can automatically discover components from an application’s classpath and create them as beans in the Spring application context. With autowiring, Spring automatically injects the components with the other beans that they depend on.

More recently, with the introduction of Spring Boot, automatic configuration has gone well beyond component scanning and autowiring. Spring Boot is an extension of the Spring Framework that offers several productivity enhancements. The most well-known of these enhancements is *autoconfiguration*, where Spring Boot can make reasonable guesses of what components need to be configured and wired together, based on entries in the classpath, environment variables, and other factors.

I’d like to show you some example code that demonstrates autoconfiguration. But I can’t. You see, autoconfiguration is much like the wind. You can see the effects of it, but there’s no code that I can show you and say “Look! Here’s an example of autoconfiguration!” Stuff happens, components are enabled, and functionality is provided without writing code. It’s this lack of code that’s essential to autoconfiguration and what makes it so wonderful.

Spring Boot autoconfiguration has dramatically reduced the amount of explicit configuration (whether with XML or Java) required to build an application. In fact, by the time you finish the example in this chapter, you’ll have a working Spring application that has only a single line of Spring configuration code!

Spring Boot enhances Spring development so much that it’s hard to imagine developing Spring applications without it. For that reason, this book treats Spring and Spring Boot as if they were one and the same. We’ll use Spring Boot as much as possible, and explicit configuration only when necessary. And, because Spring XML configuration is the old-school way of working with Spring, we’ll focus primarily on Spring’s Java-based configuration.

But enough of this chitchat, yakety-yak, and flimflam. This book’s title includes the phrase *in action*, so let’s get moving, and you can start writing your first application with Spring.

1.2 **Initializing a Spring application**

Through the course of this book, you’ll create Taco Cloud, an online application for ordering the most wonderful food created by man—tacos. Of course, you’ll use Spring, Spring Boot, and a variety of related libraries and frameworks to achieve this goal.

You’ll find several options for initializing a Spring application. Although I could walk you through the steps of manually creating a project directory structure and

defining a build specification, that's wasted time—time better spent writing application code. Therefore, you're going to lean on the Spring Initializr to bootstrap your application.

The Spring Initializr is both a browser-based web application and a REST API, which can produce a skeleton Spring project structure that you can flesh out with whatever functionality you want. Several ways to use Spring Initializr follow:

- From the web application at <http://start.spring.io>
- From the command line using the curl command
- From the command line using the Spring Boot command-line interface
- When creating a new project with Spring Tool Suite
- When creating a new project with IntelliJ IDEA
- When creating a new project with NetBeans

Rather than spend several pages of this chapter talking about each one of these options, I've collected those details in the appendix. In this chapter, and throughout this book, I'll show you how to create a new project using my favorite option: Spring Initializr support in the Spring Tool Suite.

As its name suggests, Spring Tool Suite is a fantastic Spring development environment. But it also offers a handy Spring Boot Dashboard feature that (at least at the time I write this) isn't available in any of the other IDE options.

If you're not a Spring Tool Suite user, that's fine; we can still be friends. Hop over to the appendix and substitute the Initializr option that suits you best for the instructions in the following sections. But know that throughout this book, I may occasionally reference features specific to Spring Tool Suite, such as the Spring Boot Dashboard. If you're not using Spring Tool Suite, you'll need to adapt those instructions to fit your IDE.

1.2.1 Initializing a Spring project with Spring Tool Suite

To get started with a new Spring project in Spring Tool Suite, go to the File menu and select New, and then Spring Starter Project. Figure 1.2 shows the menu structure to look for.

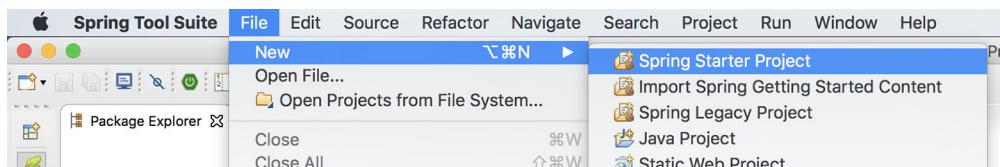


Figure 1.2 Starting a new project with the Initializr in Spring Tool Suite

Once you select Spring Starter Project, a new project wizard dialog (figure 1.3) appears. The first page in the wizard asks you for some general project information, such as the project name, description, and other essential information. If you're familiar with the

contents of a Maven pom.xml file, you’ll recognize most of the fields as items that end up in a Maven build specification. For the Taco Cloud application, fill in the dialog as shown in figure 1.3, and then click Next.

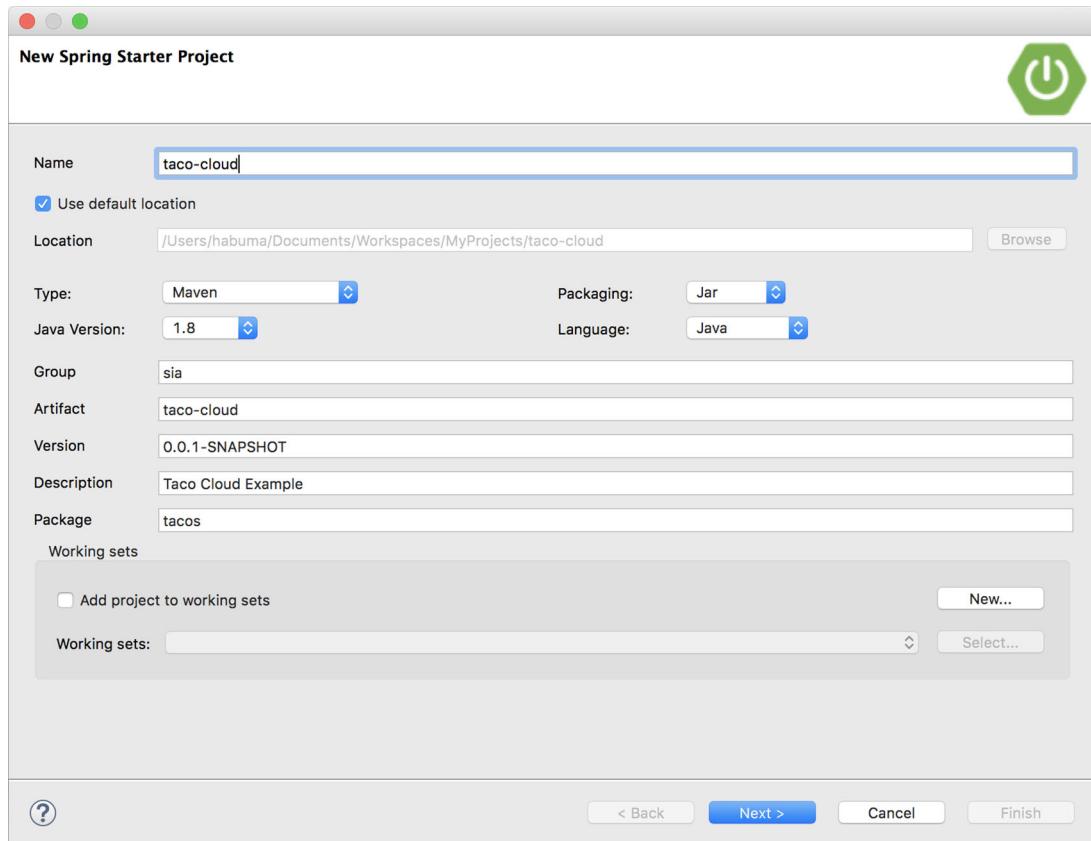


Figure 1.3 Specifying general project information for the Taco Cloud application

The next page in the wizard lets you select dependencies to add to your project (see figure 1.4). Notice that near the top of the dialog, you can select which version of Spring Boot you want to base your project on. This defaults to the most current version available. It’s generally a good idea to leave it as is unless you need to target a different version.

As for the dependencies themselves, you can either expand the various sections and seek out the desired dependencies manually, or search for them in the search box at the top of the Available list. For the Taco Cloud application, you’ll start with the dependencies shown in figure 1.4.

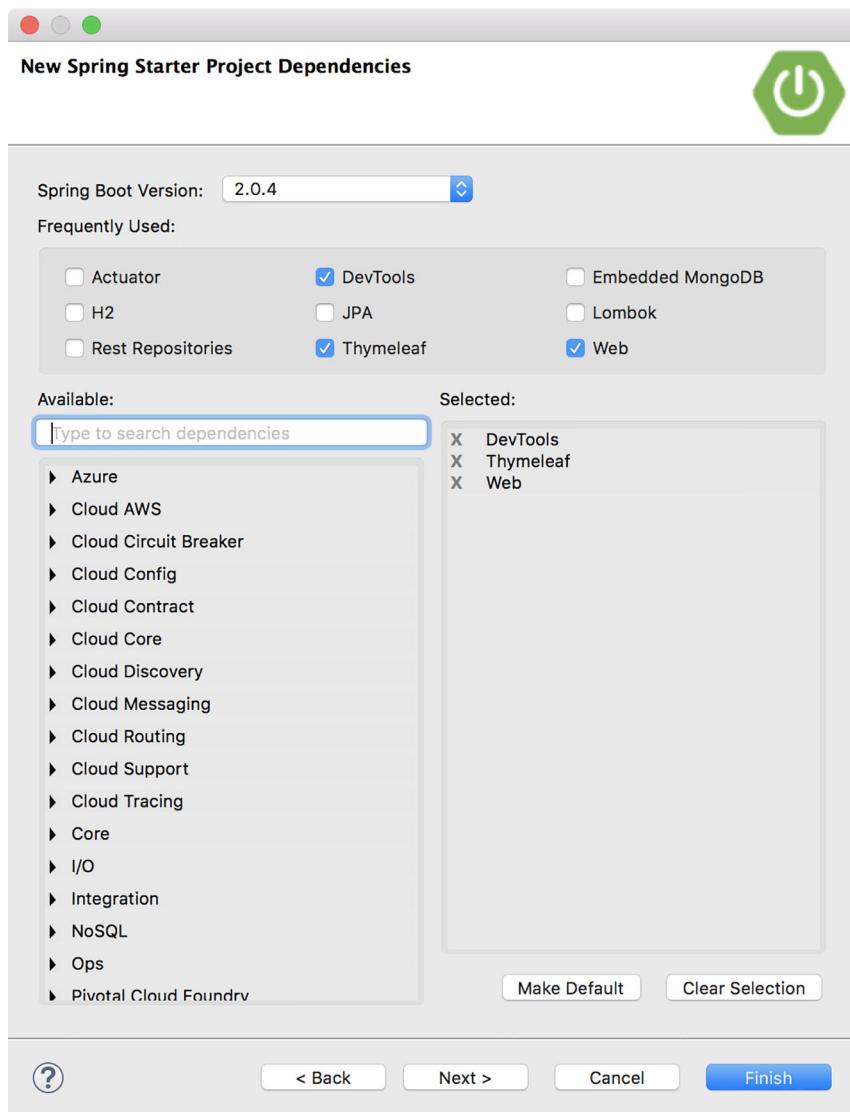


Figure 1.4 Choosing starter dependencies

At this point, you can click Finish to generate the project and add it to your workspace. But if you're feeling slightly adventurous, click Next one more time to see the final page of the new starter project wizard, as shown in figure 1.5.

By default, the new project wizard makes a call to the Spring Initializr at <http://start.spring.io> to generate the project. Generally, there's no need to override this default, which is why you could have clicked Finish on the second page of the

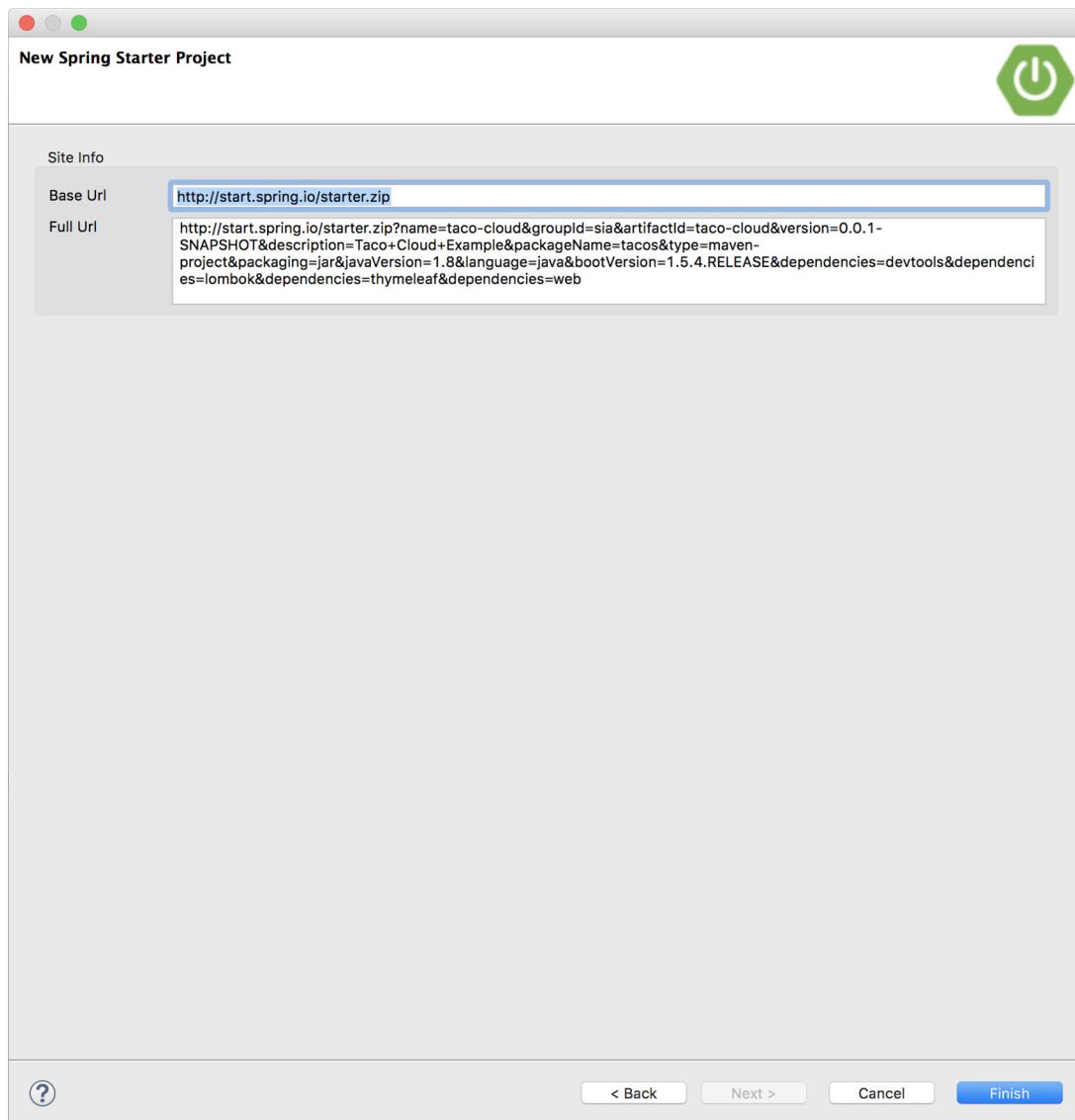


Figure 1.5 Optionally specifying an alternate Initializr address

wizard. But if for some reason you’re hosting your own clone of Initializr (perhaps a local copy on your own machine or a customized clone running inside your company firewall), then you’ll want to change the Base Url field to point to your Initializr instance before clicking Finish.

After you click Finish, the project is downloaded from the Initializr and loaded into your workspace. Wait a few moments for it to load and build, and then you’ll be

ready to start developing application functionality. But first, let's take a look at what the Initializr gave you.

1.2.2 Examining the Spring project structure

After the project loads in the IDE, expand it to see what it contains. Figure 1.6 shows the expanded Taco Cloud project in Spring Tool Suite.

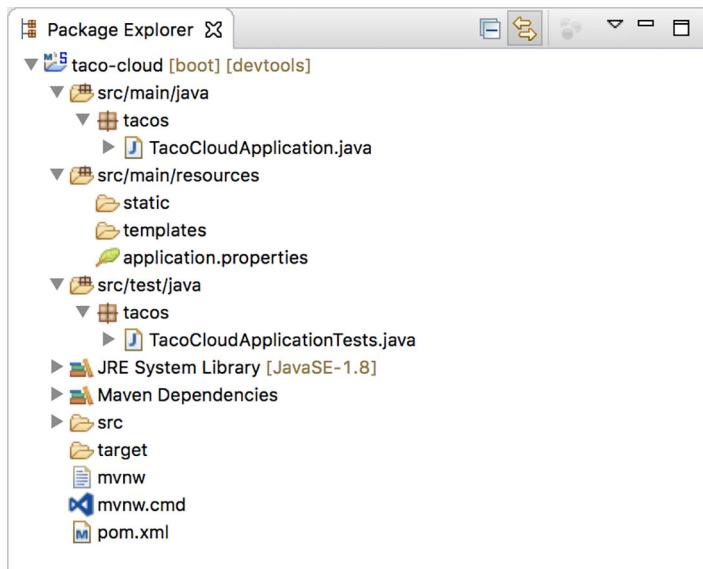


Figure 1.6 The initial Spring project structure as shown in Spring Tool Suite

You may recognize this as a typical Maven or Gradle project structure, where application source code is placed under `src/main/java`, test code is placed under `src/test/java`, and non-Java resources are placed under `src/main/resources`. Within that project structure, you'll want to take note of these items:

- `mvnw` and `mvnw.cmd`—These are Maven wrapper scripts. You can use these scripts to build your project even if you don't have Maven installed on your machine.
- `pom.xml`—This is the Maven build specification. We'll look deeper into this in a moment.
- `TacoCloudApplication.java`—This is the Spring Boot main class that bootstraps the project. We'll take a closer look at this class in a moment.
- `application.properties`—This file is initially empty, but offers a place where you can specify configuration properties. We'll tinker with this file a little in this chapter, but I'll postpone a detailed explanation of configuration properties to chapter 5.

- static—This folder is where you can place any static content (images, stylesheets, JavaScript, and so forth) that you want to serve to the browser. It's initially empty.
- templates—This folder is where you'll place template files that will be used to render content to the browser. It's initially empty, but you'll add a Thymeleaf template soon.
- TacoCloudApplicationTests.java—This is a simple test class that ensures that the Spring application context loads successfully. You'll add more tests to the mix as you develop the application.

As the Taco Cloud application grows, you'll fill in this barebones project structure with Java code, images, stylesheets, tests, and other collateral that will make your project more complete. But in the meantime, let's dig a little deeper into a few of the items that Spring Initializr provided.

EXPLORING THE BUILD SPECIFICATION

When you filled out the Initializr form, you specified that your project should be built with Maven. Therefore, the Spring Initializr gave you a pom.xml file already populated with the choices you made. The following listing shows the entire pom.xml file provided by the Initializr.

Listing 1.1 The initial Maven build specification

```
<?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>sia</groupId>
  <artifactId>taco-cloud</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>           ←—— JAR packaging

  <name>taco-cloud</name>
  <description>Taco Cloud Example</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.4.RELEASE</version>           ←—— Spring Boot version
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>
      UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>
      UTF-8</project.reporting.outputEncoding>
```

```

<java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>

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

    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>htmlunit-driver</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

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

</project>

```

The first noteworthy item in the pom.xml file is the `<packaging>` element. You chose to build your application as an executable JAR file, as opposed to a WAR file. This is probably one of the most curious choices you'll make, especially for a web application. After all, traditional Java web applications are packaged as WAR files, leaving JAR files the packaging of choice for libraries and the occasional desktop UI application.

The choice of JAR packaging is a cloud-minded choice. Whereas WAR files are perfectly suitable for deploying to a traditional Java application server, they’re not a natural fit for most cloud platforms. Although some cloud platforms (such as Cloud Foundry) are capable of deploying and running WAR files, all Java cloud platforms are capable of running an executable JAR file. Therefore, the Spring Initializr defaults to JAR packaging unless you tell it to do otherwise.

If you intend to deploy your application to a traditional Java application server, then you’ll need to choose WAR packaging and include a web initializer class. We’ll look at how to build WAR files in more detail in chapter 2.

Next, take note of the `<parent>` element and, more specifically, its `<version>` child. This specifies that your project has `spring-boot-starter-parent` as its parent POM. Among other things, this parent POM provides dependency management for several libraries commonly used in Spring projects. For those libraries covered by the parent POM, you won’t have to specify a version, as it’s inherited from the parent. The version, `2.0.4.RELEASE`, indicates that you’re using Spring Boot 2.0.4 and, thus, will inherit dependency management as defined by that version of Spring Boot.

While we’re on the subject of dependencies, note that there are three dependencies declared under the `<dependencies>` element. The first two should look somewhat familiar to you. They correspond directly to the Web and Thymeleaf dependencies that you selected before clicking the Finish button in the Spring Tool Suite new project wizard. The third dependency is one that provides a lot of helpful testing capabilities. You didn’t have to check a box for it to be included because the Spring Initializr assumes (hopefully, correctly) that you’ll be writing tests.

You may also notice that all three dependencies have the word *starter* in their artifact ID. Spring Boot starter dependencies are special in that they typically don’t have any library code themselves, but instead transitively pull in other libraries. These starter dependencies offer three primary benefits:

- Your build file will be significantly smaller and easier to manage because you won’t need to declare a dependency on every library you might need.
- You’re able to think of your dependencies in terms of what capabilities they provide, rather than in terms of library names. If you’re developing a web application, you’ll add the web starter dependency rather than a laundry list of individual libraries that enable you to write a web application.
- You’re freed from the burden of worry about library versions. You can trust that for a given version of Spring Boot, the versions of the libraries brought in transitively will be compatible. You only need to worry about which version of Spring Boot you’re using.

Finally, the build specification ends with the Spring Boot plugin. This plugin performs a few important functions:

- It provides a Maven goal that enables you to run the application using Maven. You’ll try out this goal in section 1.3.4.

- It ensures that all dependency libraries are included within the executable JAR file and available on the runtime classpath.
- It produces a manifest file in the JAR file that denotes the bootstrap class (`TacoCloudApplication`, in your case) as the main class for the executable JAR.

Speaking of the bootstrap class, let's open it up and take a closer look.

BOOTSTRAPPING THE APPLICATION

Because you'll be running the application from an executable JAR, it's important to have a main class that will be executed when that JAR file is run. You'll also need at least a minimal amount of Spring configuration to bootstrap the application. That's what you'll find in the `TacoCloudApplication` class, shown in the following listing.

Listing 1.2 The Taco Cloud bootstrap class

```
package tacos;

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

@SpringBootApplication
public class TacoCloudApplication { ← Spring Boot application

    public static void main(String[] args) {
        SpringApplication.run(TacoCloudApplication.class, args); ← Runs the application
    }
}
```

Although there's little code in `TacoCloudApplication`, what's there packs quite a punch. One of the most powerful lines of code is also one of the shortest. The `@SpringBootApplication` annotation clearly signifies that this is a Spring Boot application. But there's more to `@SpringBootApplication` than meets the eye.

`@SpringBootApplication` is a composite annotation that combines three other annotations:

- `@SpringBootConfiguration`—Designates this class as a configuration class. Although there's not much configuration in the class yet, you can add Java-based Spring Framework configuration to this class if you need to. This annotation is, in fact, a specialized form of the `@Configuration` annotation.
- `@EnableAutoConfiguration`—Enables Spring Boot automatic configuration. We'll talk more about autoconfiguration later. For now, know that this annotation tells Spring Boot to automatically configure any components that it thinks you'll need.
- `@ComponentScan`—Enables component scanning. This lets you declare other classes with annotations like `@Component`, `@Controller`, `@Service`, and others, to have Spring automatically discover them and register them as components in the Spring application context.

The other important piece of `TacoCloudApplication` is the `main()` method. This is the method that will be run when the JAR file is executed. For the most part, this method is boilerplate code; every Spring Boot application you write will have a method similar or identical to this one (class name differences notwithstanding).

The `main()` method calls a static `run()` method on the `SpringApplication` class, which performs the actual bootstrapping of the application, creating the Spring application context. The two parameters passed to the `run()` method are a configuration class and the command-line arguments. Although it's not necessary that the configuration class passed to `run()` be the same as the bootstrap class, this is the most convenient and typical choice.

Chances are you won't need to change anything in the bootstrap class. For simple applications, you might find it convenient to configure one or two other components in the bootstrap class, but for most applications, you're better off creating a separate configuration class for anything that isn't autoconfigured. You'll define several configuration classes throughout the course of this book, so stay tuned for details.

TESTING THE APPLICATION

Testing is an important part of software development. Recognizing this, the Spring Initializr gives you a test class to get started. The following listing shows the baseline test class.

Listing 1.3 A baseline application test

```
package tacos;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class TacoCloudApplicationTests {

    @Test
    public void contextLoads() {
    }
}
```

There's not much to be seen in `TacoCloudApplicationTests`: the one test method in the class is empty. Even so, this test class does perform an essential check to ensure that the Spring application context can be loaded successfully. If you make any changes that prevent the Spring application context from being created, this test fails, and you can react by fixing the problem.

Also notice the class annotated with `@RunWith(SpringRunner.class)`. `@RunWith` is a JUnit annotation, providing a test runner that guides JUnit in running a test. Think

of it as applying a plugin to JUnit to provide custom testing behavior. In this case, JUnit is given `SpringRunner`, a Spring-provided test runner that provides for the creation of a Spring application context that the test will run against.

A TEST RUNNER BY ANY OTHER NAME...

If you're already familiar with writing Spring tests or are maybe looking at some existing Spring-based test classes, you may have seen a test runner named `SpringJUnit4ClassRunner`. `SpringRunner` is an alias for `SpringJUnit4ClassRunner`, and was introduced in Spring 4.3 to remove the association with a specific version of JUnit (for example, JUnit 4). And there's no denying that the alias is easier to read and type.

`@SpringBootTest` tells JUnit to bootstrap the test with Spring Boot capabilities. For now, it's enough to think of this as the test class equivalent of calling `SpringApplication.run()` in a `main()` method. Over the course of this book, you'll see `@SpringBootTest` several times, and we'll uncover some of its power.

Finally, there's the test method itself. Although `@RunWith(SpringRunner.class)` and `@SpringBootTest` are tasked to load the Spring application context for the test, they won't have anything to do if there aren't any test methods. Even without any assertions or code of any kind, this empty test method will prompt the two annotations to do their job and load the Spring application context. If there are any problems in doing so, the test fails.

At this point, we've concluded our review of the code provided by the Spring Initializr. You've seen some of the boilerplate foundation that you can use to develop a Spring application, but you still haven't written a single line of code. Now it's time to fire up your IDE, dust off your keyboard, and add some custom code to the Taco Cloud application.

1.3 Writing a Spring application

Because you're just getting started, we'll start off with a relatively small change to the Taco Cloud application, but one that will demonstrate a lot of Spring's goodness. It seems appropriate that as you're just starting, the first feature you'll add to the Taco Cloud application is a homepage. As you add the homepage, you'll create two code artifacts:

- A controller class that handles requests for the homepage
- A view template that defines what the homepage looks like

And because testing is important, you'll also write a simple test class to test the homepage. But first things first ... let's write that controller.

1.3.1 Handling web requests

Spring comes with a powerful web framework known as Spring MVC. At the center of Spring MVC is the concept of a *controller*, a class that handles requests and responds with information of some sort. In the case of a browser-facing application, a controller

responds by optionally populating model data and passing the request on to a view to produce HTML that's returned to the browser.

You're going to learn a lot about Spring MVC in chapter 2. But for now, you'll write a simple controller class that handles requests for the root path (for example, /) and forwards those requests to the homepage view without populating any model data. The following listing shows the simple controller class.

Listing 1.4 The homepage controller

```
package tacos;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller           ← The controller
public class HomeController {

    @GetMapping("/")      ← Handles requests
    public String home() { ← for the root path /
        return "home";     ← Returns the
    }                      ← view name
}
```

As you can see, this class is annotated with `@Controller`. On its own, `@Controller` doesn't do much. Its primary purpose is to identify this class as a component for component scanning. Because `HomeController` is annotated with `@Controller`, Spring's component scanning automatically discovers it and creates an instance of `HomeController` as a bean in the Spring application context.

In fact, a handful of other annotations (including `@Component`, `@Service`, and `@Repository`) serve a purpose similar to `@Controller`. You could have just as effectively annotated `HomeController` with any of those other annotations, and it would have still worked the same. The choice of `@Controller` is, however, more descriptive of this component's role in the application.

The `home()` method is as simple as controller methods come. It's annotated with `@GetMapping` to indicate that if an HTTP GET request is received for the root path /, then this method should handle that request. It does so by doing nothing more than returning a `String` value of `home`.

This value is interpreted as the logical name of a view. How that view is implemented depends on a few factors, but because Thymeleaf is in your classpath, you can define that template with Thymeleaf.

WHY THYMELEAF?

You may be wondering why you chose Thymeleaf for a template engine. Why not JSP? Why not FreeMarker? Why not one of several other options?

Put simply, I had to choose something, and I like Thymeleaf and generally prefer it over those other options. And even though JSP may seem like an obvious choice,

there are some challenges to overcome when using JSP with Spring Boot. I didn't want to go down that rabbit hole in chapter 1. Hang tight. We'll look at other template options, including JSP, in chapter 2.

The template name is derived from the logical view name by prefixing it with /templates/ and postfixing it with .html. The resulting path for the template is /templates/home.html. Therefore, you'll need to place the template in your project at /src/main/resources/templates/home.html. Let's create that template now.

1.3.2 Defining the view

In the interest of keeping your homepage simple, it should do nothing more than welcome users to the site. The next listing shows the basic Thymeleaf template that defines the Taco Cloud homepage.

Listing 1.5 The Taco Cloud homepage template

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
  </head>

  <body>
    <h1>Welcome to...</h1>
    
  </body>
</html>
```

There's not much to discuss with regard to this template. The only notable line of code is the one with the `` tag to display the Taco Cloud logo. It uses a Thymeleaf `th:src` attribute and an `@{...}` expression to reference the image with a context-relative path. Aside from that, it's not much more than a Hello World page.

But let's talk about that image a bit more. I'll leave it up to you to define a Taco Cloud logo that you like. You'll need to make sure you place it at the right place within the project.

The image is referenced with the context-relative path `/images/TacoCloud.png`. As you'll recall from our review of the project structure, static content such as images is kept in the `/src/main/resources/static` folder. That means that the Taco Cloud logo image must also reside within the project at `/src/main/resources/static/images/TacoCloud.png`.

Now that you've got a controller to handle requests for the homepage and a view template to render the homepage, you're almost ready to fire up the application and see it in action. But first, let's see how you can write a test against the controller.

1.3.3 Testing the controller

Testing web applications can be tricky when making assertions against the content of an HTML page. Fortunately, Spring comes with some powerful test support that makes testing a web application easy.

For the purposes of the homepage, you'll write a test that's comparable in complexity to the homepage itself. Your test will perform an HTTP GET request for the root path / and expect a successful result where the view name is home and the resulting content contains the phrase "Welcome to...". The following should do the trick.

Listing 1.6 A test for the homepage controller

```
package tacos;

import static org.hamcrest.Matchers.containsString;
import static
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest(HomeController.class)
public class HomeControllerTest {    ← Web test for HomeController

    @Autowired
    private MockMvc mockMvc;    ← Injects MockMvc

    @Test
    public void testHomePage() throws Exception {
        mockMvc.perform(get("/"))
            ← Performs GET /
            .andExpect(status().isOk())
            ← Expects HTTP 200
            .andExpect(view().name("home"))
            ← Expects home view
            .andExpect(content().string(
                containsString("Welcome to...")));
    }    ← Expects Welcome to...
}
```

The first thing you might notice about this test is that it differs slightly from the `TacoCloudApplicationTests` class with regard to the annotations applied to it. Instead of `@SpringBootTest` markup, `HomeControllerTest` is annotated with `@WebMvcTest`. This is a special test annotation provided by Spring Boot that arranges for the test to run in the context of a Spring MVC application. More specifically, in this case, it arranges for `HomeController` to be registered in Spring MVC so that you can throw requests against it.

`@WebMvcTest` also sets up Spring support for testing Spring MVC. Although it could be made to start a server, mocking the mechanics of Spring MVC is sufficient for your purposes. The test class is injected with a `MockMvc` object for the test to drive the mockup.

The `testHomePage()` method defines the test you want to perform against the homepage. It starts with the `MockMvc` object to perform an HTTP GET request for `/` (the root path). From that request, it sets the following expectations:

- The response should have an HTTP 200 (OK) status.
- The view should have a logical name of `home`.
- The rendered view should contain the text “Welcome to....”

If, after the `MockMvc` object performs the request, any of those expectations aren’t met, then the test fails. But your controller and view template are written to satisfy those expectations, so the test should pass with flying colors—or at least with some shade of green indicating a passing test.

The controller has been written, the view template created, and you have a passing test. It seems that you’ve implemented the homepage successfully. But even though the test passes, there’s something slightly more satisfying with seeing the results in a browser. After all, that’s how Taco Cloud customers are going to see it. Let’s build the application and run it.

1.3.4 Building and running the application

Just as there are several ways to initialize a Spring application, there are several ways to run one. If you like, you can flip over to the appendix to read about some of the more common ways to run a Spring Boot application.

Because you chose to use Spring Tool Suite to initialize and work on the project, you have a handy feature called the Spring Boot Dashboard available to help you run your application inside the IDE. The Spring Boot Dashboard appears as a tab, typically near the bottom left of the IDE window. Figure 1.7 shows an annotated screenshot of the Spring Boot Dashboard.

I don’t want to spend much time going over everything the Spring Boot Dashboard does, although figure 1.7 covers some of the most useful details. The important thing to know right now is how to use it to run the Taco Cloud application. Make sure `taco-cloud` application is highlighted in the list of projects (it’s the only application shown in figure 1.7), and then click the start button (the left-most button with both a green triangle and a red square). The application should start right up.

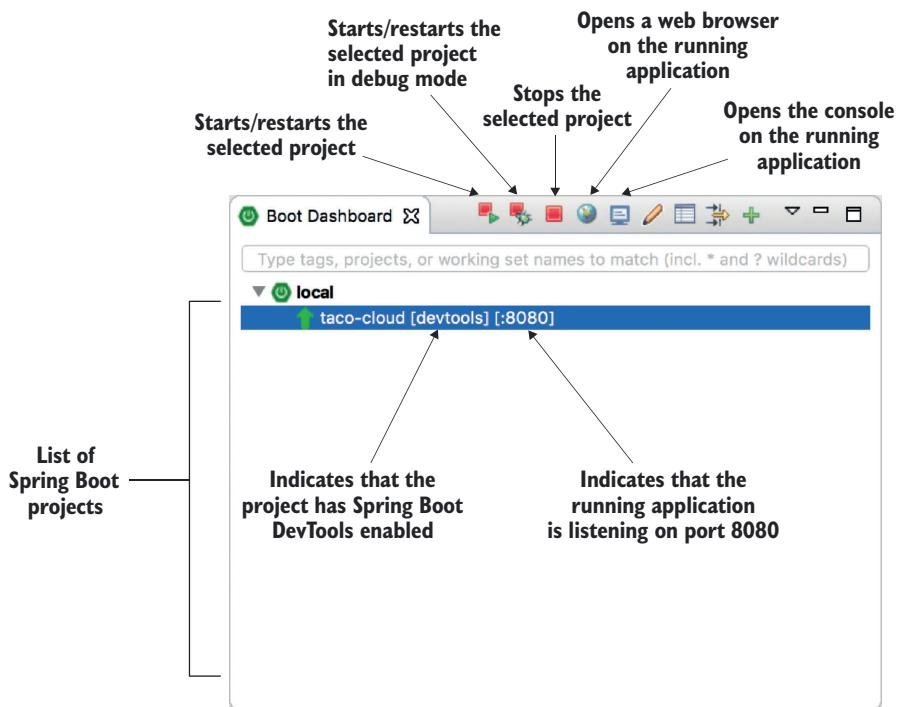


Figure 1.7 Highlights of the Spring Boot Dashboard

As the application starts, you'll see some Spring ASCII art fly by in the console, followed by some log entries describing the steps as the application starts. Before the logging stops, you'll see a log entry saying Tomcat started on port(s): 8080 (http), which means that you're ready to point your web browser at the homepage to see the fruits of your labor.

Wait a minute. Tomcat started? When did you deploy the application to Tomcat?

Spring Boot applications tend to bring everything they need with them and don't need to be deployed to some application server. You never deployed your application to Tomcat ... Tomcat is a part of your application! (I'll describe the details of how Tomcat became part of your application in section 1.3.6.)

Now that the application has started, point your web browser to `http://localhost:8080` (or click the globe button in the Spring Boot Dashboard) and you should see something like figure 1.8. Your results may be different if you designed your own logo image. But it shouldn't vary much from what you see in figure 1.8.

It may not be much to look at. But this isn't exactly a book on graphic design. The humble appearance of the homepage is more than sufficient for now. And it provides you a solid start on getting to know Spring.

One thing I've glossed over until now is DevTools. You selected it as a dependency when initializing your project. It appears as a dependency in the produced

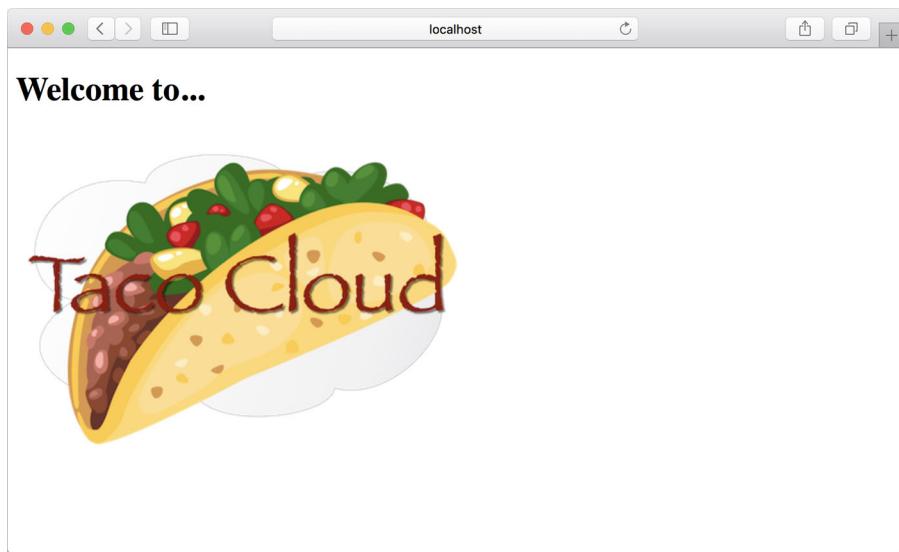


Figure 1.8 The Taco Cloud homepage

pom.xml file. And the Spring Boot Dashboard even shows that the project has DevTools enabled. But what is DevTools, and what does it do for you? Let's take a quick survey of a couple of DevTools' most useful features.

1.3.5 Getting to know Spring Boot DevTools

As its name suggests, DevTools provides Spring developers with some handy development-time tools. Among those are

- Automatic application restart when code changes
- Automatic browser refresh when browser-destined resources (such as templates, JavaScript, stylesheets, and so on) change
- Automatic disable of template caches
- Built in H2 Console if the H2 database is in use

It's important to understand that DevTools isn't an IDE plugin, nor does it require that you use a specific IDE. It works equally well in Spring Tool Suite, IntelliJ IDEA, and NetBeans. Furthermore, because it's only intended for development purposes, it's smart enough to disable itself when deploying in a production setting. (We'll discuss how it does this when you get around to deploying your application in chapter 19.) For now, let's focus on the most useful features of Spring Boot DevTools, starting with automatic application restart.

AUTOMATIC APPLICATION RESTART

With DevTools as part of your project, you'll be able to make changes to Java code and properties files in the project and see those changes applied after a brief moment.

DevTools monitors for changes, and when it sees something has changed, it automatically restarts the application.

More precisely, when DevTools is in play, the application is loaded into two separate class loaders in the Java virtual machine (JVM). One class loader is loaded with your Java code, property files, and pretty much anything that's in the `src/main/` path of the project. These are items that are likely to change frequently. The other class loader is loaded with dependency libraries, which aren't likely to change as often.

When a change is detected, DevTools reloads only the class loader containing your project code and restarts the Spring application context, but leaves the other class loader and the JVM intact. Although subtle, this strategy affords a small reduction in the time it takes to start the application.

The downside of this strategy is that changes to dependencies won't be available in automatic restarts. That's because the class loader containing dependency libraries isn't automatically reloaded. This means that any time you add, change, or remove a dependency in your build specification, you'll need to do a hard restart of the application for those changes to take effect.

AUTOMATIC BROWSER REFRESH AND TEMPLATE CACHE DISABLE

By default, template options such as Thymeleaf and FreeMarker are configured to cache the results of template parsing so that templates don't need to be reparsed with every request they serve. This is great in production, as it buys a bit of performance benefit.

Cached templates, however, are not so great at development time. Cached templates make it impossible to make changes to the templates while the application is running and see the results after refreshing the browser. Even if you've made changes, the cached template will still be in use until you restart the application.

DevTools addresses this issue by automatically disabling all template caching. Make as many changes as you want to your templates and know that you're only a browser refresh away from seeing the results.

But if you're like me, you don't even want to be burdened with the effort of clicking the browser's refresh button. It'd be much nicer if you could make the changes and witness the results in the browser immediately. Fortunately, DevTools has something special for those of us who are too lazy to click a refresh button.

When DevTools is in play, it automatically enables a LiveReload (<http://livereload.com/>) server along with your application. By itself, the LiveReload server isn't very useful. But when coupled with a corresponding LiveReload browser plugin, it causes your browser to automatically refresh when changes are made to templates, images, stylesheets, JavaScript, and so on—in fact, almost anything that ends up being served to your browser.

LiveReload has browser plugins for Google Chrome, Safari, and Firefox browsers. (Sorry, Internet Explorer and Edge fans.) Visit <http://livereload.com/extensions/> to find information on how to install LiveReload for your browser.

BUILT IN H2 CONSOLE

Although your project doesn't yet use a database, that will change in chapter 3. If you choose to use the H2 database for development, DevTools will also automatically enable an H2 Console that you can access from your web browser. You only need to point your web browser to <http://localhost:8080/h2-console> to gain insight into the data your application is working with.

At this point, you've written a complete, albeit simple, Spring application. You'll expand on it throughout the course of the book. But now is a good time to step back and review what you've accomplished and how Spring played a part.

1.3.6 Let's review

Think back on how you got to this point. In short, these are the steps you've taken to build your Spring-based Taco Cloud application:

- You created an initial project structure using Spring Initializr.
- You wrote a controller class to handle the homepage request.
- You defined a view template to render the homepage.
- You wrote a simple test class to prove out your work.

Seems pretty straightforward, doesn't it? With the exception of the first step to bootstrap the project, each action you've taken has been keenly focused on achieving the goal of producing a homepage.

In fact, almost every line of code you've written is aimed toward that goal. Not counting Java `import` statements, I count only two lines of code in your controller class and no lines in the view template that are Spring-specific. And although the bulk of the test class utilizes Spring testing support, it seems a little less invasive in the context of a test.

That's an important benefit of developing with Spring. You can focus on the code that meets the requirements of an application rather than on satisfying the demands of a framework. Although you'll no doubt need to write some framework-specific code from time to time, it'll usually be only a small fraction of your codebase. As I said before, Spring (with Spring Boot) can be considered the *frameworkless framework*.

How does this even work? What is Spring doing behind the scenes to make sure your application needs are met? To understand what Spring is doing, let's start by looking at the build specification.

In the `pom.xml` file, you declared a dependency on the Web and Thymeleaf starters. These two dependencies transitively brought in a handful of other dependencies, including

- Spring's MVC framework
- Embedded Tomcat
- Thymeleaf and the Thymeleaf layout dialect

It also brought Spring Boot’s autoconfiguration library along for the ride. When the application starts, Spring Boot autoconfiguration detects those libraries and automatically

- Configures the beans in the Spring application context to enable Spring MVC
- Configures the embedded Tomcat server in the Spring application context
- Configures a Thymeleaf view resolver for rendering Spring MVC views with Thymeleaf templates

In short, autoconfiguration does all the grunt work, leaving you to focus on writing code that implements your application functionality. That’s a pretty sweet arrangement, if you ask me!

Your Spring journey has just begun. The Taco Cloud application only touched on a small portion of what Spring has to offer. Before you take your next step, let’s survey the Spring landscape and see what landmarks you’ll encounter on your journey.

1.4 ***Surveying the Spring landscape***

To get an idea of the Spring landscape, look no further than the enormous list of checkboxes on the full version of the Spring Initializr web form. It lists over 100 dependency choices, so I won’t try to list them all here or to provide a screenshot. But I encourage you to take a look. In the meantime, I’ll mention a few of the highlights.

1.4.1 ***The core Spring Framework***

As you might expect, the core Spring Framework is the foundation of everything else in the Spring universe. It provides the core container and dependency injection framework. But it also provides a few other essential features.

Among these is Spring MVC, Spring’s web framework. You’ve already seen how to use Spring MVC to write a controller class to handle web requests. What you’ve not yet seen, however, is that Spring MVC can also be used to create REST APIs that produce non-HTML output. We’re going to dig more into Spring MVC in chapter 2 and then take another look at how to use it to create REST APIs in chapter 6.

The core Spring Framework also offers some elemental data persistence support, specifically template-based JDBC support. You’ll see how to use `JdbcTemplate` in chapter 3.

In the most recent version of Spring (5.0.8), support was added for reactive-style programming, including a new reactive web framework called Spring WebFlux that borrows heavily from Spring MVC. You’ll look at Spring’s reactive programming model in part 3 and Spring WebFlux specifically in chapter 10.

1.4.2 ***Spring Boot***

We’ve already seen many of the benefits of Spring Boot, including starter dependencies and autoconfiguration. Be certain that we’ll use as much of Spring Boot as possible throughout this book and avoid any form of explicit configuration, unless it’s

absolutely necessary. But in addition to starter dependencies and autoconfiguration, Spring Boot also offers a handful of other useful features:

- The Actuator provides runtime insight into the inner workings of an application, including metrics, thread dump information, application health, and environment properties available to the application.
- Flexible specification of environment properties.
- Additional testing support on top of the testing assistance found in the core framework.

What's more, Spring Boot offers an alternative programming model based on Groovy scripts that's called the Spring Boot CLI (command-line interface). With the Spring Boot CLI, you can write entire applications as a collection of Groovy scripts and run them from the command line. We won't spend much time with the Spring Boot CLI, but we'll touch on it on occasion when it fits our needs.

Spring Boot has become such an integral part of Spring development; I can't imagine developing a Spring application without it. Consequently, this book takes a Spring Boot-centric view, and you might catch me using the word *Spring* when I'm referring to something that Spring Boot is doing.

1.4.3 Spring Data

Although the core Spring Framework comes with basic data persistence support, Spring Data provides something quite amazing: the ability to define your application's data repositories as simple Java interfaces, using a naming convention when defining methods to drive how data is stored and retrieved.

What's more, Spring Data is capable of working with a several different kinds of databases, including relational (JPA), document (Mongo), graph (Neo4j), and others. You'll use Spring Data to help create repositories for the Taco Cloud application in chapter 3.

1.4.4 Spring Security

Application security has always been an important topic, and it seems to become more important every day. Fortunately, Spring has a robust security framework in Spring Security.

Spring Security addresses a broad range of application security needs, including authentication, authorization, and API security. Although the scope of Spring Security is too large to be properly covered in this book, we'll touch on some of the most common use cases in chapters 4 and 12.

1.4.5 Spring Integration and Spring Batch

At some point, most applications will need to integrate with other applications or even with other components of the same application. Several patterns of application

integration have emerged to address these needs. Spring Integration and Spring Batch provide the implementation of these patterns for Spring-based applications.

Spring Integration addresses real-time integration where data is processed as it's made available. In contrast, Spring Batch addresses batched integration where data is allowed to collect for a time until some trigger (perhaps a time trigger) signals that it's time for the batch of data to be processed. You'll explore both Spring Batch and Spring Integration in chapter 9.

1.4.6 **Spring Cloud**

As I'm writing this, the application development world is entering a new era where we'll no longer develop our applications as single deployment unit monoliths and will instead compose applications from several individual deployment units known as *microservices*.

Microservices are a hot topic, addressing several practical development and runtime concerns. In doing so, however, they bring to fore their own challenges. Those challenges are met head-on by Spring Cloud, a collection of projects for developing cloud-native applications with Spring.

Spring Cloud covers a lot of ground, and it'd be impossible to cover it all in this book. We'll look at some of the most common components of Spring Cloud in chapters 13, 14, and 15. For a more complete discussion of Spring Cloud, I suggest taking a look at *Spring Microservices in Action* by John Carnell (Manning, 2017, www.manning.com/books/spring-microservices-in-action).

Summary

- Spring aims to make developer challenges easy, like creating web applications, working with databases, securing applications, and microservices.
- Spring Boot builds on top of Spring to make Spring even easier with simplified dependency management, automatic configuration, and runtime insights.
- Spring applications can be initialized using the Spring Initializr, which is web-based and supported natively in most Java development environments.
- The components, commonly referred to as beans, in a Spring application context can be declared explicitly with Java or XML, discovered by component scanning, or automatically configured with Spring Boot autoconfiguration.



Developing web applications

This chapter covers

- Presenting model data in the browser
- Processing and validating form input
- Choosing a view template library

First impressions are important. Curb appeal can sell a house long before the home buyer enters the door. A car’s cherry paint job will turn more heads than what’s under the hood. And literature is replete with stories of love at first sight. What’s inside is very important, but what’s outside—what’s seen first—is important.

The applications you’ll build with Spring will do all kinds of things, including crunching data, reading information from a database, and interacting with other applications. But the first impression your application users will get comes from the user interface. And in many applications, that UI is a web application presented in a browser.

In chapter 1, you created your first Spring MVC controller to display your application homepage. But Spring MVC can do far more than simply display static content. In this chapter, you’ll develop the first major bit of functionality in your Taco Cloud application—the ability to design custom tacos. In doing so, you’ll dig deeper into Spring MVC, and you’ll see how to display model data and process form input.

2.1 Displaying information

Fundamentally, Taco Cloud is a place where you can order tacos online. But more than that, Taco Cloud wants to enable its customers to express their creative side and to design custom tacos from a rich palette of ingredients.

Therefore, the Taco Cloud web application needs a page that displays the selection of ingredients for taco artists to choose from. The ingredient choices may change at any time, so they shouldn't be hardcoded into an HTML page. Rather, the list of available ingredients should be fetched from a database and handed over to the page to be displayed to the customer.

In a Spring web application, it's a controller's job to fetch and process data. And it's a view's job to render that data into HTML that will be displayed in the browser. You're going to create the following components in support of the taco creation page:

- A domain class that defines the properties of a taco ingredient
- A Spring MVC controller class that fetches ingredient information and passes it along to the view
- A view template that renders a list of ingredients in the user's browser

The relationship between these components is illustrated in figure 2.1.

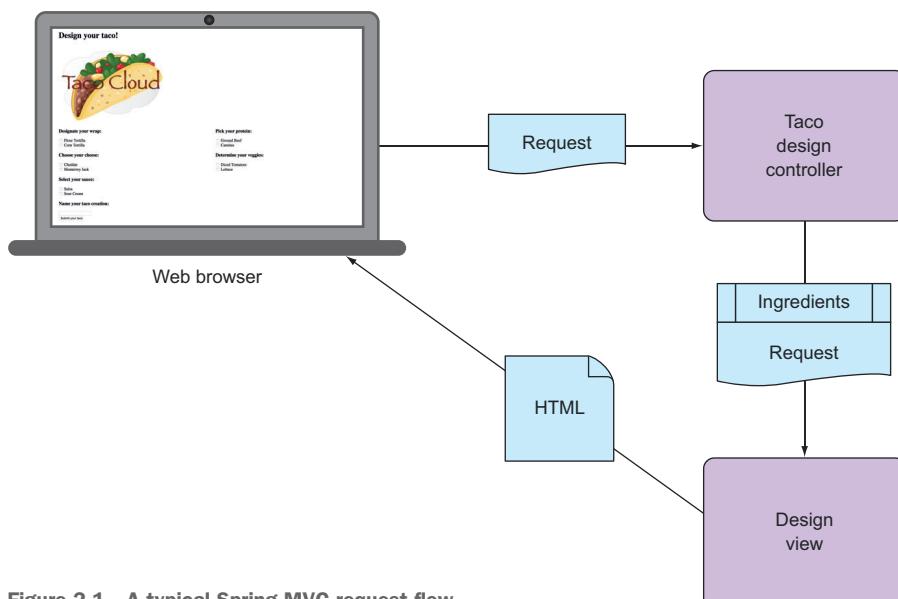


Figure 2.1 A typical Spring MVC request flow

Because this chapter focuses on Spring's web framework, we'll defer any of the database stuff to chapter 3. For now, the controller will be solely responsible for providing the ingredients to the view. In chapter 3, you'll rework the controller to collaborate with a repository that fetches ingredients data from a database.

Before you write the controller and view, let's hammer out the domain type that represents an ingredient. This will establish a foundation on which you can develop your web components.

2.1.1 Establishing the domain

An application's domain is the subject area that it addresses—the ideas and concepts that influence the understanding of the application.¹ In the Taco Cloud application, the domain includes such objects as taco designs, the ingredients that those designs are composed of, customers, and taco orders placed by the customers. To get started, we'll focus on taco ingredients.

In your domain, taco ingredients are fairly simple objects. Each has a name as well as a type so that it can be visually categorized (proteins, cheeses, sauces, and so on). Each also has an ID by which it can easily and unambiguously be referenced. The following `Ingredient` class defines the domain object you need.

Listing 2.1 Defining taco ingredients

```
package tacos;

import lombok.Data;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
public class Ingredient {

    private final String id;
    private final String name;
    private final Type type;

    public static enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }

}
```

As you can see, this is a run-of-the-mill Java domain class, defining the three properties needed to describe an ingredient. Perhaps the most unusual thing about the `Ingredient` class as defined in listing 2.1 is that it seems to be missing the usual set of getter and setter methods, not to mention useful methods like `equals()`, `hashCode()`, `toString()`, and others.

You don't see them in the listing partly to save space, but also because you're using an amazing library called Lombok to automatically generate those methods at runtime. In fact, the `@Data` annotation at the class level is provided by Lombok and tells

¹ For a much more in-depth discussion of application domains, I suggest Eric Evans' *Domain-Driven Design* (Addison-Wesley Professional, 2003).

Lombok to generate all of those missing methods as well as a constructor that accepts all `final` properties as arguments. By using Lombok, you can keep the code for `Ingredient` slim and trim.

Lombok isn't a Spring library, but it's so incredibly useful that I find it hard to develop without it. And it's a lifesaver when I need to keep code examples in a book short and sweet.

To use Lombok, you'll need to add it as a dependency in your project. If you're using Spring Tool Suite, it's an easy matter of right-clicking on the `pom.xml` file and selecting Edit Starters from the Spring context menu option. The same selection of dependencies you were given in chapter 1 (in figure 1.4) will appear, giving you a chance to add or change your selected dependencies. Find the Lombok choice, make sure it's checked, and click OK; Spring Tool Suite will automatically add it to your build specification.

Alternatively, you can manually add it with the following entry in `pom.xml`:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

This dependency will provide you with Lombok annotations (such as `@Data`) at development time and with automatic method generation at runtime. But you'll also need to add Lombok as an extension in your IDE, or your IDE will complain with errors about missing methods and `final` properties that aren't being set. Visit <https://projectlombok.org/> to find out how to install Lombok in your IDE of choice.

I think you'll find Lombok to be very useful, but know that it's optional. You don't need it to develop Spring applications, so if you'd rather not use it, feel free to write those missing methods by hand. Go ahead ... I'll wait. When you finish, you'll add some controllers to handle web requests in your application.

2.1.2 **Creating a controller class**

Controllers are the major players in Spring's MVC framework. Their primary job is to handle HTTP requests and either hand a request off to a view to render HTML (browser-displayed) or write data directly to the body of a response (RESTful). In this chapter, we're focusing on the kinds of controllers that use views to produce content for web browsers. When we get to chapter 6, we'll look at writing controllers that handle requests in a REST API.

For the Taco Cloud application, you need a simple controller that will do the following:

- Handle HTTP GET requests where the request path is `/design`
- Build a list of ingredients
- Hand the request and the ingredient data off to a view template to be rendered as HTML and sent to the requesting web browser

The following `DesignTacoController` class addresses those requirements.

Listing 2.2 The beginnings of a Spring controller class

```
package tacos.web;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import lombok.extern.slf4j.Slf4j;
import tacos.Taco;
import tacos.Ingredient;
import tacos.Ingredient.Type;

@Slf4j
@Controller
@RequestMapping("/design")
public class DesignTacoController {

    @GetMapping
    public String showDesignForm(Model model) {
        List<Ingredient> ingredients = Arrays.asList(
            new Ingredient("FLTO", "Flour Tortilla", Type.WRAP),
            new Ingredient("COTO", "Corn Tortilla", Type.WRAP),
            new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),
            new Ingredient("CARN", "Carnitas", Type.PROTEIN),
            new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES),
            new Ingredient("LETC", "Lettuce", Type.VEGGIES),
            new Ingredient("CHED", "Cheddar", Type.CHEESE),
            new Ingredient("JACK", "Monterrey Jack", Type.CHEESE),
            new Ingredient("SLSA", "Salsa", Type.SAUCE),
            new Ingredient("SRCR", "Sour Cream", Type.SAUCE)
        );

        Type[] types = Ingredient.Type.values();
        for (Type type : types) {
            model.addAttribute(type.toString().toLowerCase(),
                filterByType(ingredients, type));
        }

        model.addAttribute("design", new Taco());
        return "design";
    }
}
```

The first thing to note about `DesignTacoController` is the set of annotations applied at the class level. The first, `@Slf4j`, is a Lombok-provided annotation that, at runtime, will automatically generate an SLF4J (Simple Logging Facade for Java, <https://www.slf4j.org/>) Logger in the class. This modest annotation has the same effect as if you were to explicitly add the following lines within the class:

```
private static final org.slf4j.Logger log =
    org.slf4j.LoggerFactory.getLogger(DesignTacoController.class);
```

You'll make use of this Logger a little later.

The next annotation applied to `DesignTacoController` is `@Controller`. This annotation serves to identify this class as a controller and to mark it as a candidate for component scanning, so that Spring will discover it and automatically create an instance of `DesignTacoController` as a bean in the Spring application context.

`DesignTacoController` is also annotated with `@RequestMapping`. The `@RequestMapping` annotation, when applied at the class level, specifies the kind of requests that this controller handles. In this case, it specifies that `DesignTacoController` will handle requests whose path begins with `/design`.

HANDLING A GET REQUEST

The class-level `@RequestMapping` specification is refined with the `@GetMapping` annotation that adorns the `showDesignForm()` method. `@GetMapping`, paired with the class-level `@RequestMapping`, specifies that when an HTTP GET request is received for `/design`, `showDesignForm()` will be called to handle the request.

`@GetMapping` is a relatively new annotation, having been introduced in Spring 4.3. Prior to Spring 4.3, you might have used a method-level `@RequestMapping` annotation instead:

```
@RequestMapping(method=RequestMethod.GET)
```

Clearly, `@GetMapping` is more succinct and specific to the HTTP method that it targets. `@GetMapping` is just one member of a family of request-mapping annotations. Table 2.1 lists all of the request-mapping annotations available in Spring MVC.

Table 2.1 Spring MVC request-mapping annotations

Annotation	Description
<code>@RequestMapping</code>	General-purpose request handling
<code>@GetMapping</code>	Handles HTTP GET requests
<code>@PostMapping</code>	Handles HTTP POST requests
<code>@PutMapping</code>	Handles HTTP PUT requests
<code>@DeleteMapping</code>	Handles HTTP DELETE requests
<code>@PatchMapping</code>	Handles HTTP PATCH requests

Making the right thing the easy thing

It's always a good idea to be as specific as possible when declaring request mappings on your controller methods. At the very least, this means declaring both a path (or inheriting a path from the class-level `@RequestMapping`) and which HTTP method it will handle.

The lengthier `@RequestMapping(method=RequestMethod.GET)` made it tempting to take the lazy way out and leave off the `method` attribute. Thanks to Spring 4.3's new mapping annotations, the right thing to do is also the easy thing to do—with less typing.

The new request-mapping annotations have all of the same attributes as `@RequestMapping`, so you can use them anywhere you'd otherwise use `@RequestMapping`.

Generally, I prefer to only use `@RequestMapping` at the class level to specify the base path. I use the more specific `@GetMapping`, `@PostMapping`, and so on, on each of the handler methods.

Now that you know that the `showDesignForm()` method will handle the request, let's look at the method body to see how it ticks. The bulk of the method constructs a list of `Ingredient` objects. The list is hardcoded for now. When we get to chapter 3, you'll pull the list of available taco ingredients from a database.

Once the list of ingredients is ready, the next few lines of `showDesignForm()` filters the list by ingredient type. A list of ingredient types is then added as an attribute to the `Model` object that's passed into `showDesignForm()`. `Model` is an object that ferries data between a controller and whatever view is charged with rendering that data. Ultimately, data that's placed in `Model` attributes is copied into the servlet response attributes, where the view can find them. The `showDesignForm()` method concludes by returning "design", which is the logical name of the view that will be used to render the model to the browser.

Your `DesignTacoController` is really starting to take shape. If you were to run the application now and point your browser at the `/design` path, the `DesignTacoController`'s `showDesignForm()` would be engaged, fetching data from the repository and placing it in the model before passing the request on to the view. But because you haven't defined the view yet, the request would take a horrible turn, resulting in an HTTP 404 (Not Found) error. To fix that, let's switch our attention to the view where the data will be decorated with HTML to be presented in the user's web browser.

2.1.3 Designing the view

After the controller is finished with its work, it's time for the view to get going. Spring offers several great options for defining views, including JavaServer Pages (JSP), Thymeleaf, FreeMarker, Mustache, and Groovy-based templates. For now, we'll use Thymeleaf, the choice we made in chapter 1 when starting the project. We'll consider a few of the other options in section 2.5.

In order to use Thymeleaf, you need to add another dependency to your project build. The following `<dependency>` entry uses Spring Boot's Thymeleaf starter to make Thymeleaf available for rendering the view you're about to create:

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

At runtime, Spring Boot autoconfiguration will see that Thymeleaf is in the classpath and will automatically create the beans that support Thymeleaf views for Spring MVC.

View libraries such as Thymeleaf are designed to be decoupled from any particular web framework. As such, they're unaware of Spring's model abstraction and are unable to work with the data that the controller places in `Model`. But they can work with servlet request attributes. Therefore, before Spring hands the request over to a view, it copies the model data into request attributes that Thymeleaf and other view templating options have ready access to.

Thymeleaf templates are just HTML with some additional element attributes that guide a template in rendering request data. For example, if there were a request attribute whose key is `"message"`, and you wanted it to be rendered into an HTML `<p>` tag by Thymeleaf, you'd write the following in your Thymeleaf template:

```
<p th:text="${message}">placeholder message</p>
```

When the template is rendered into HTML, the body of the `<p>` element will be replaced with the value of the servlet request attribute whose key is `"message"`. The `th:text` attribute is a Thymeleaf-namespaced attribute that performs the replacement. The `${}` operator tells it to use the value of a request attribute (`"message"`, in this case).

Thymeleaf also offers another attribute, `th:each`, that iterates over a collection of elements, rendering the HTML once for each item in the collection. This will come in handy as you design your view to list taco ingredients from the model. For example, to render just the list of `"wrap"` ingredients, you can use the following snippet of HTML:

```
<h3>Designate your wrap:</h3>
<div th:each="ingredient : ${wrap}">
  <input name="ingredients" type="checkbox" th:value="${ingredient.id}" />
  <span th:text="${ingredient.name}">INGREDIENT</span><br/>
</div>
```

Here, you use the `th:each` attribute on the `<div>` tag to repeat rendering of the `<div>` once for each item in the collection found in the `wrap` request attribute. On each iteration, the ingredient item is bound to a Thymeleaf variable named `ingredient`.

Inside the `<div>` element, there's a check box `<input>` element and a `` element to provide a label for the check box. The check box uses Thymeleaf's `th:value` to set the rendered `<input>` element's `value` attribute to the value found in the

ingredient's `id` property. The `` element uses `th:text` to replace the "INGREDIENT" placeholder text with the value of the ingredient's `name` property.

When rendered with actual model data, one iteration of that `<div>` loop might look like this:

```
<div>
    <input name="ingredients" type="checkbox" value="FLTO" />
    <span>Flour Tortilla</span><br/>
</div>
```

Ultimately, the preceding Thymeleaf snippet is just part of a larger HTML form through which your taco artist users will submit their tasty creations. The complete Thymeleaf template, including all ingredient types and the form, is shown in the following listing.

Listing 2.3 The complete design-a-taco page

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Taco Cloud</title>
        <link rel="stylesheet" th:href="@{/styles.css}" />
    </head>

    <body>
        <h1>Design your taco!</h1>
        

        <form method="POST" th:object="${design}">
            <div class="grid">
                <div class="ingredient-group" id="wraps">
                    <h3>Designate your wrap:</h3>
                    <div th:each="ingredient : ${wrap}">
                        <input name="ingredients" type="checkbox" th:value="${ingredient.id}" />
                        <span th:text="${ingredient.name}">INGREDIENT</span><br/>
                    </div>
                </div>
            </div>

            <div class="ingredient-group" id="proteins">
                <h3>Pick your protein:</h3>
                <div th:each="ingredient : ${protein}">
                    <input name="ingredients" type="checkbox" th:value="${ingredient.id}" />
                    <span th:text="${ingredient.name}">INGREDIENT</span><br/>
                </div>
            </div>

            <div class="ingredient-group" id="cheeses">
                <h3>Choose your cheese:</h3>
                <div th:each="ingredient : ${cheese}">
```

```

<input name="ingredients" type="checkbox" th:value="${ingredient.id}"
/>
<span th:text="${ingredient.name}">INGREDIENT</span><br/>
</div>
</div>

<div class="ingredient-group" id="veggies">
<h3>Determine your veggies:</h3>
<div th:each="ingredient : ${veggies}">
<input name="ingredients" type="checkbox" th:value="${ingredient.id}"
/>
<span th:text="${ingredient.name}">INGREDIENT</span><br/>
</div>
</div>

<div class="ingredient-group" id="sauces">
<h3>Select your sauce:</h3>
<div th:each="ingredient : ${sauce}">
<input name="ingredients" type="checkbox" th:value="${ingredient.id}"
/>
<span th:text="${ingredient.name}">INGREDIENT</span><br/>
</div>
</div>
</div>

<div>

<h3>Name your taco creation:</h3>
<input type="text" th:field="*{name}" />
<br/>

<button>Submit your taco</button>
</div>
</form>
</body>
</html>

```

As you can see, you repeat the `<div>` snippet for each of the types of ingredients. And you include a Submit button and field where the user can name their creation.

It's also worth noting that the complete template includes the Taco Cloud logo image and a `<link>` reference to a stylesheet.² In both cases, Thymeleaf's `@{}` operator is used to produce a context-relative path to the static artifacts that they're referencing. As you learned in chapter 1, static content in a Spring Boot application is served from the `/static` directory at the root of the classpath.

Now that your controller and view are complete, you can fire up the application to see the fruits of your labor. There are many ways to run a Spring Boot application. In chapter 1, I showed you how to run the application by first building it into an executable

² The contents of the stylesheet aren't relevant to our discussion; it only contains styling to present the ingredients in two columns instead of one long list of ingredients.

JAR file and then running the JAR with `java -jar`. I also showed how you can run the application directly from the build with `mvn spring-boot:run`.

No matter how you fire up the Taco Cloud application, once it starts, point your browser to <http://localhost:8080/design>. You should see a page that looks something like figure 2.2.

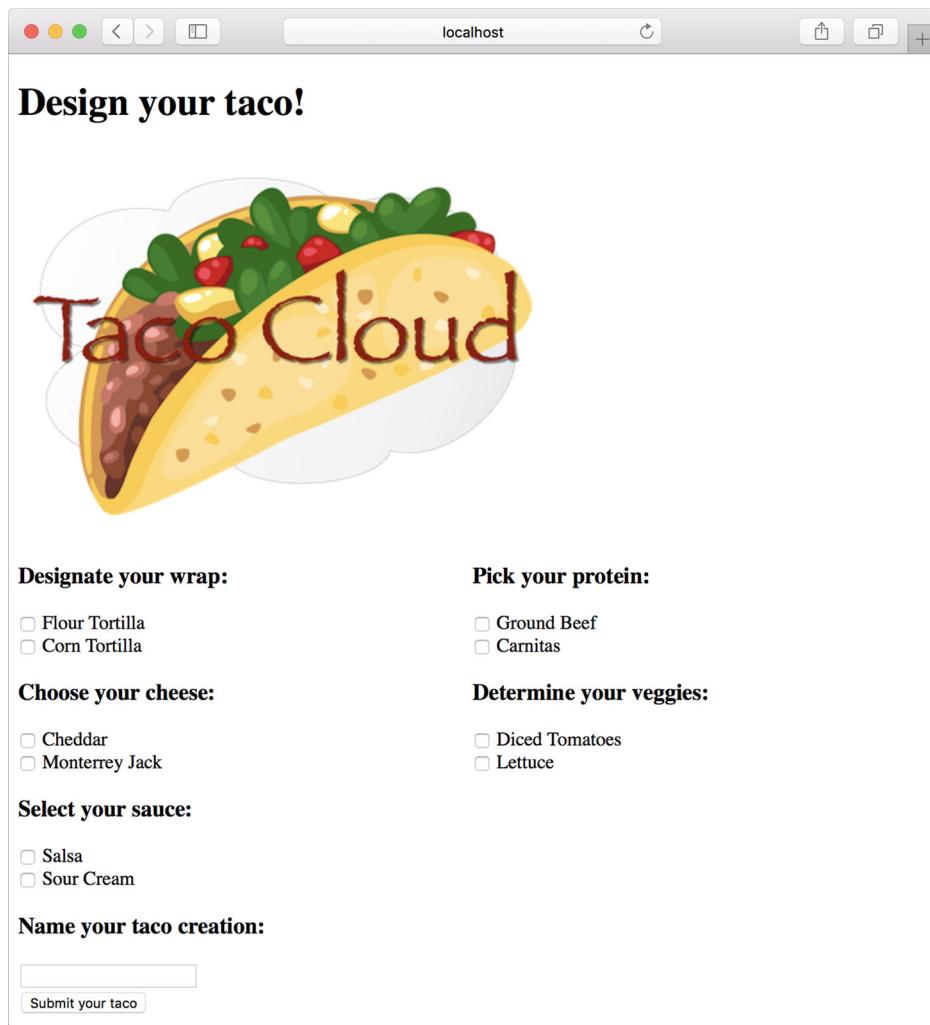


Figure 2.2 The rendered taco design page

It's looking good! A taco artist visiting your site is presented with a form containing a palette of taco ingredients from which they can create their masterpiece. But what happens when they click the Submit Your Taco button?

Your DesignTacoController isn't yet ready to accept taco creations. If the design form is submitted, the user will be presented with an error. (Specifically, it will be an HTTP 405 error: Request Method “POST” Not Supported.) Let's fix that by writing some more controller code that handles form submission.

2.2 Processing form submission

If you take another look at the `<form>` tag in your view, you can see that its `method` attribute is set to `POST`. Moreover, the `<form>` doesn't declare an `action` attribute. This means that when the form is submitted, the browser will gather up all the data in the form and send it to the server in an HTTP POST request to the same path for which a GET request displayed the form—the `/design` path.

Therefore, you need a controller handler method on the receiving end of that POST request. You need to write a new handler method in `DesignTacoController` that handles a POST request for `/design`.

In listing 2.2, you used the `@GetMapping` annotation to specify that the `showDesignForm()` method should handle HTTP GET requests for `/design`. Just like `@GetMapping` handles GET requests, you can use `@PostMapping` to handle POST requests. For handling taco design submissions, add the `processDesign()` method in the following listing to `DesignTacoController`.

Listing 2.4 Handling POST requests with `@PostMapping`

```
@PostMapping
public String processDesign(Design design) {
    // Save the taco design...
    // We'll do this in chapter 3
    log.info("Processing design: " + design);

    return "redirect:/orders/current";
}
```

As applied to the `processDesign()` method, `@PostMapping` coordinates with the class-level `@RequestMapping` to indicate that `processDesign()` should handle POST requests for `/design`. This is precisely what you need to process a taco artist's submitted creations.

When the form is submitted, the fields in the form are bound to properties of a Taco object (whose class is shown in the next listing) that's passed as a parameter into `processDesign()`. From there, the `processDesign()` method can do whatever it wants with the Taco object.

Listing 2.5 A domain object defining a taco design

```
package tacos;
import java.util.List;
import lombok.Data;
```

```
@Data  
public class Taco {  
  
    private String name;  
    private List<String> ingredients;  
  
}
```

As you can see, Taco is a straightforward Java domain object with a couple of properties. Like Ingredient, the Taco class is annotated with @Data to automatically generate essential JavaBean methods for you at runtime.

If you look back at the form in listing 2.3, you'll see several checkbox elements, all with the name ingredients, and a text input element named name. Those fields in the form correspond directly to the ingredients and name properties of the Taco class.

The Name field on the form only needs to capture a simple textual value. Thus the name property of Taco is of type String. The ingredients check boxes also have textual values, but because zero or many of them may be selected, the ingredients property that they're bound to is a List<String> that will capture each of the chosen ingredients.

For now, the processDesign() method does nothing with the Taco object. In fact, it doesn't do much of anything at all. That's OK. In chapter 3, you'll add some persistence logic that will save the submitted Taco to a database.

Just as with the showDesignForm() method, processDesign() finishes by returning a String value. And just like showDesignForm(), the value returned indicates a view that will be shown to the user. But what's different is that the value returned from processDesign() is prefixed with "redirect:", indicating that this is a redirect view. More specifically, it indicates that after processDesign() completes, the user's browser should be redirected to the relative path /order/current.

The idea is that after creating a taco, the user will be redirected to an order form from which they can place an order to have their taco creations delivered. But you don't yet have a controller that will handle a request for /orders/current.

Given what you now know about @Controller, @RequestMapping, and @GetMapping, you can easily create such a controller. It might look something like the following listing.

Listing 2.6 A controller to present a taco order form

```
package tacos.web;  
import javax.validation.Valid;  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;  
import org.springframework.validation.Errors;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
import lombok.extern.slf4j.Slf4j;  
import tacos.Order;
```

```

@Slf4j
@Controller
@RequestMapping("/orders")
public class OrderController {

    @GetMapping("/current")
    public String orderForm(Model model) {
        model.addAttribute("order", new Order());
        return "orderForm";
    }

}

```

Once again, you use Lombok's `@Slf4j` annotation to create a free SLF4J Logger object at runtime. You'll use this Logger in a moment to log the details of the order that's submitted.

The class-level `@RequestMapping` specifies that any request-handling methods in this controller will handle requests whose path begins with `/orders`. When combined with the method-level `@GetMapping`, it specifies that the `orderForm()` method will handle HTTP GET requests for `/orders/current`.

As for the `orderForm()` method itself, it's extremely basic, only returning a logical view name of `orderForm`. Once you have a way to persist taco creations to a database in chapter 3, you'll revisit this method and modify it to populate the model with a list of Taco objects to be placed in the order.

The `orderForm` view is provided by a Thymeleaf template named `orderForm.html`, which is shown next.

Listing 2.7 A taco order form view

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Taco Cloud</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
</head>

<body>

    <form method="POST" th:action="@{/orders}" th:object="${order}">
        <h1>Order your taco creations!</h1>

        
        <a th:href="@{/design}" id="another">Design another taco</a><br/>

        <div th:if="#{#fields.hasErrors()}">
            <span class="validationError">
                Please correct the problems below and resubmit.
            </span>
        </div>
    </form>

```

```
<h3>Deliver my taco masterpieces to...</h3>
<label for="name">Name: </label>
<input type="text" th:field="*{name}" />
<br/>

<label for="street">Street address: </label>
<input type="text" th:field="*{street}" />
<br/>

<label for="city">City: </label>
<input type="text" th:field="*{city}" />
<br/>

<label for="state">State: </label>
<input type="text" th:field="*{state}" />
<br/>

<label for="zip">Zip code: </label>
<input type="text" th:field="*{zip}" />
<br/>

<h3>Here's how I'll pay...</h3>
<label for="ccNumber">Credit Card #: </label>
<input type="text" th:field="*{ccNumber}" />
<br/>

<label for="ccExpiration">Expiration: </label>
<input type="text" th:field="*{ccExpiration}" />
<br/>

<label for="ccCVV">CVV: </label>
<input type="text" th:field="*{ccCVV}" />
<br/>

<input type="submit" value="Submit order"/>
</form>

</body>
</html>
```

For the most part, the `orderForm.html` view is typical HTML/Thymeleaf content, with very little of note. But notice that the `<form>` tag here is different from the `<form>` tag used in listing 2.3 in that it also specifies a form action. Without an action specified, the form would submit an HTTP POST request back to the same URL that presented the form. But here, you specify that the form should be POSTed to `/orders` (using Thymeleaf's `@{...}` operator for a context-relative path).

Therefore, you're going to need to add another method to your `OrderController` class that handles POST requests for `/orders`. You won't have a way to persist orders until the next chapter, so you'll keep it simple here—something like what you see in the next listing.

Listing 2.8 Handling a taco order submission

```
@PostMapping
public String processOrder(Order order) {
    log.info("Order submitted: " + order);
    return "redirect:/";
}
```

When the `processOrder()` method is called to handle a submitted order, it's given an `Order` object whose properties are bound to the submitted form fields. `Order`, much like `Taco`, is a fairly straightforward class that carries order information.

Listing 2.9 A domain object for taco orders

```
package tacos;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.NotBlank;
import lombok.Data;

@Data
public class Order {

    private String name;
    private String street;
    private String city;
    private String state;
    private String zip;
    private String ccNumber;
    private String ccExpiration;
    private String ccCVV;
}

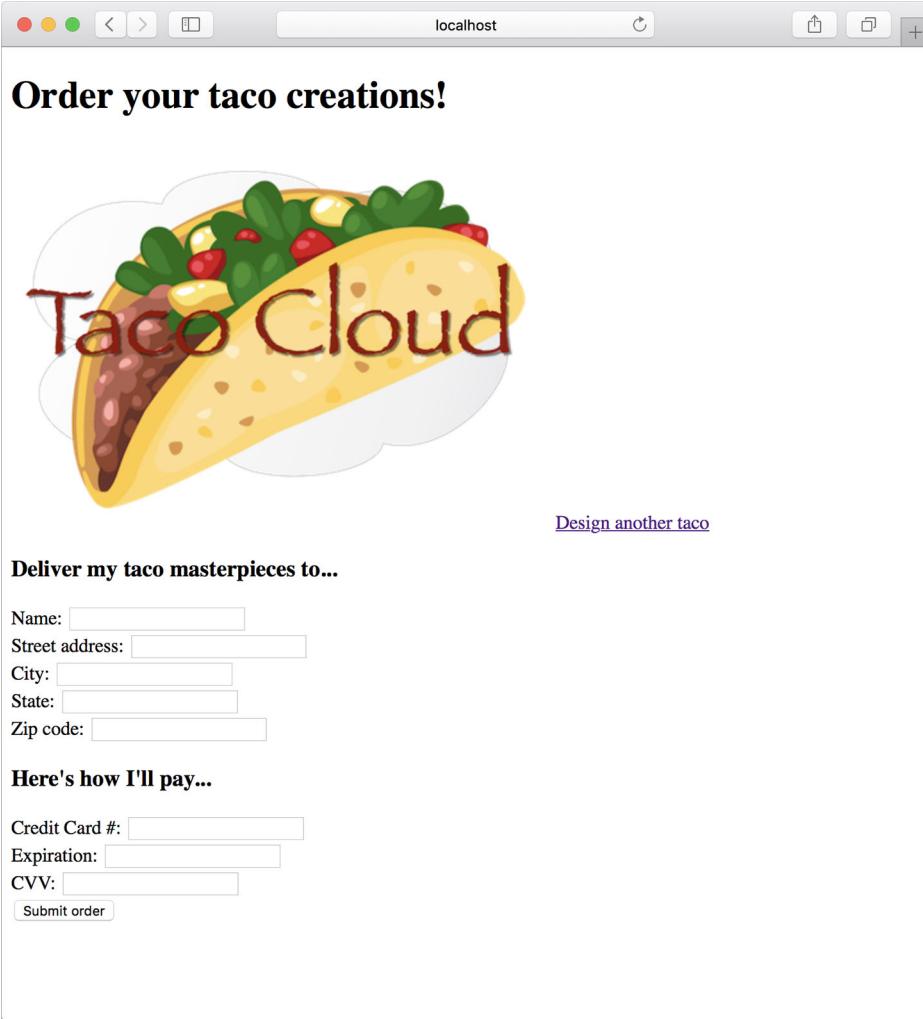
}
```

Now that you've developed an `OrderController` and the order form view, you're ready to try it out. Open your browser to <http://localhost:8080/design>, select some ingredients for your taco, and click the Submit Your Taco button. You should see a form similar to what's shown in figure 2.3.

Fill in some fields in the form, and press the Submit Order button. As you do, keep an eye on the application logs to see your order information. When I tried it, the log entry looked something like this (reformatted to fit the width of this page):

```
Order submitted: Order(name=Craig Walls,street1=1234 7th Street,
                      city=Somewhere, state=Who knows?, zip=zipzap, ccNumber=Who can guess?,
                      ccExpiration=Some day, ccCVV=See-vee-vee)
```

If you look carefully at the log entry from my test order, you can see that although the `processOrder()` method did its job and handled the form submission, it let a little bit of bad information get in. Most of the fields in the form contained data that couldn't



The screenshot shows a web browser window with the URL "localhost" in the address bar. The main content area has a title "Order your taco creations!" in bold black font. Below the title is a large, stylized illustration of a taco filled with meat, cheese, and vegetables, with the words "Taco Cloud" written across it in a red, hand-drawn style. At the bottom right of the page is a link "Design another taco".

Deliver my taco masterpieces to...

Name:

Street address:

City:

State:

Zip code:

Here's how I'll pay...

Credit Card #:

Expiration:

CVV:

Figure 2.3 The taco order form

possibly be correct. Let's add some validation to ensure that the data provided at least resembles the kind of information required.

2.3 Validating form input

When designing a new taco creation, what if the user selects no ingredients or fails to specify a name for their creation? When submitting the order, what if they fail to fill in the required address fields? Or what if they enter a value into the credit card field that isn't even a valid credit card number?

As things stand now, nothing will stop the user from creating a taco without any ingredients or with an empty delivery address, or even submitting the lyrics to their

favorite song as the credit card number. That's because you haven't yet specified how those fields should be validated.

One way to perform form validation is to litter the `processDesign()` and `processOrder()` methods with a bunch of `if/then` blocks, checking each and every field to ensure that it meets the appropriate validation rules. But that would be cumbersome and difficult to read and debug.

Fortunately, Spring supports Java's Bean Validation API (also known as JSR-303; <https://jcp.org/en/jsr/detail?id=303>). This makes it easy to declare validation rules as opposed to explicitly writing declaration logic in your application code. And with Spring Boot, you don't need to do anything special to add validation libraries to your project, because the Validation API and the Hibernate implementation of the Validation API are automatically added to the project as transient dependencies of Spring Boot's web starter.

To apply validation in Spring MVC, you need to

- Declare validation rules on the class that is to be validated: specifically, the `Taco` class.
- Specify that validation should be performed in the controller methods that require validation: specifically, the `DesignTacoController`'s `processDesign()` method and `OrderController`'s `processOrder()` method.
- Modify the form views to display validation errors.

The Validation API offers several annotations that can be placed on properties of domain objects to declare validation rules. Hibernate's implementation of the Validation API adds even more validation annotations. Let's see how you can apply a few of these annotations to validate a submitted `Taco` or `Order`.

2.3.1 Declaring validation rules

For the `Taco` class, you want to ensure that the `name` property isn't empty or null and that the list of selected ingredients has at least one item. The following listing shows an updated `Taco` class that uses `@NotNull` and `@Size` to declare those validation rules.

Listing 2.10 Adding validation to the `Taco` domain class

```
package tacos;
import java.util.List;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import lombok.Data;

@Data
public class Taco {

    @NotNull
    @Size(min=5, message="Name must be at least 5 characters long")
    private String name;
```

```
@Size(min=1, message="You must choose at least 1 ingredient")
private List<String> ingredients;

}
```

You'll notice that in addition to requiring that the name property isn't null, you declare that it should have a value that's at least 5 characters in length.

When it comes to declaring validation on submitted taco orders, you must apply annotations to the Order class. For the address properties, you only want to be sure that the user doesn't leave any of the fields blank. For that, you'll use Hibernate Validator's @NotBlank annotation.

Validation of the payment fields, however, is a bit more exotic. You need to not only ensure that the ccNumber property isn't empty, but that it contains a value that could be a valid credit card number. The ccExpiration property must conform to a format of MM/YY (two-digit month and year). And the ccCVV property needs to be a three-digit number. To achieve this kind of validation, you need to use a few other Java Bean Validation API annotations and borrow a validation annotation from the Hibernate Validator collection of annotations. The following listing shows the changes needed to validate the Order class.

Listing 2.11 Validating order fields

```
package tacos;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import javax.validation.constraints.NotBlank;
import lombok.Data;

@Data
public class Order {

    @NotBlank(message="Name is required")
    private String name;

    @NotBlank(message="Street is required")
    private String street;

    @NotBlank(message="City is required")
    private String city;

    @NotBlank(message="State is required")
    private String state;

    @NotBlank(message="Zip code is required")
    private String zip;

    @CreditCardNumber(message="Not a valid credit card number")
    private String ccNumber;
```

```

@Pattern(regexp="^(0[1-9]|1[0-2])([\\/])([1-9][0-9])$",
         message="Must be formatted MM/YY")
private String ccExpiration;

@Digits(integer=3, fraction=0, message="Invalid CVV")
private String ccCVV;

}

```

As you can see, the `ccNumber` property is annotated with `@CreditCardNumber`. This annotation declares that the property's value must be a valid credit card number that passes the Luhn algorithm check (https://en.wikipedia.org/wiki/Luhn_algorithm). This prevents user mistakes and deliberately bad data but doesn't guarantee that the credit card number is actually assigned to an account or that the account can be used for charging.

Unfortunately, there's no ready-made annotation for validating the MM/YY format of the `ccExpiration` property. I've applied the `@Pattern` annotation, providing it with a regular expression that ensures that the property value adheres to the desired format. If you're wondering how to decipher the regular expression, I encourage you to check out the many online regular expression guides, including <http://www.regular-expressions.info/>. Regular expression syntax is a dark art and certainly outside the scope of this book.

Finally, the `ccCVV` property is annotated with `@Digits` to ensure that the value contains exactly three numeric digits.

All of the validation annotations include a `message` attribute that defines the message you'll display to the user if the information they enter doesn't meet the requirements of the declared validation rules.

2.3.2 **Performing validation at form binding**

Now that you've declared how a `Taco` and `Order` should be validated, we need to revisit each of the controllers, specifying that validation should be performed when the forms are POSTed to their respective handler methods.

To validate a submitted `Taco`, you need to add the Java Bean Validation API's `@Valid` annotation to the `Taco` argument of `DesignTacoController`'s `processDesign()` method.

Listing 2.12 Validating a POSTed Taco

```

@PostMapping
public String processDesign(@Valid Taco design, Errors errors) {
    if (errors.hasErrors()) {
        return "design";
    }

    // Save the taco design...
    // We'll do this in chapter 3
    log.info("Processing design: " + design);
}

```

```
    return "redirect:/orders/current";
}
```

The `@Valid` annotation tells Spring MVC to perform validation on the submitted Taco object after it's bound to the submitted form data and before the `processDesign()` method is called. If there are any validation errors, the details of those errors will be captured in an `Errors` object that's passed into `processDesign()`. The first few lines of `processDesign()` consult the `Errors` object, asking its `hasErrors()` method if there are any validation errors. If there are, the method concludes without processing the Taco and returns the "design" view name so that the form is redisplayed.

To perform validation on submitted Order objects, similar changes are also required in the `processOrder()` method of `OrderController`.

Listing 2.13 Validating a POSTed Order

```
@PostMapping
public String processOrder(@Valid Order order, Errors errors) {
    if (errors.hasErrors()) {
        return "orderForm";
    }

    log.info("Order submitted: " + order);
    return "redirect:/";
}
```

In both cases, the method will be allowed to process the submitted data if there are no validation errors. If there are validation errors, the request will be forwarded to the form view to give the user a chance to correct their mistakes.

But how will the user know what mistakes require correction? Unless you call out the errors on the form, the user will be left guessing about how to successfully submit the form.

2.3.3 Displaying validation errors

Thymeleaf offers convenient access to the `Errors` object via the `fields` property and with its `th:errors` attribute. For example, to display validation errors on the credit card number field, you can add a `` element that uses these error references to the order form template, as follows.

Listing 2.14 Displaying validation errors

```
<label for="ccNumber">Credit Card #: </label>
<input type="text" th:field="*{ccNumber}" />
<span class="validationError"
      th:if="#{fields.hasErrors('ccNumber')}"
      th:errors="*{ccNumber}">CC Num Error</span>
```

Aside from a `class` attribute that can be used to style the error so that it catches the user's attention, the `` element uses a `th:if` attribute to decide whether or not

to display the ``. The `fields` property's `hasErrors()` method checks if there are any errors in the `ccNumber` field. If so, the `` will be rendered.

The `th:errors` attribute references the `ccNumber` field and, assuming there are errors for that field, it will replace the placeholder content of the `` element with the validation message.

If you were to sprinkle similar `` tags around the order form for the other fields, you might see a form that looks like figure 2.4 when you submit invalid information. The errors indicate that the name, city, and ZIP code fields have been left blank, and that all of the payment fields fail to meet the validation criteria.

The screenshot shows a web browser window with the title bar "localhost". The main content area displays a large graphic of a taco filled with meat, cheese, and vegetables, with the text "Taco Cloud" overlaid. Below the graphic, a message says "Please correct the problems below and resubmit." To the right is a link "Design another taco". The form fields include:

- Name: Name is required
- Street address: 1234 7th Street
- City: City is required
- State: VT
- Zip code: Zip code is required

Below the form, a section titled "Here's how I'll pay..." contains:

- Credit Card #: Who can guess? Not a valid credit card number
- Expiration: Some day Must be formatted MM/YY
- CVV: See-vee-vee Invalid CVV

A "Submit order" button is at the bottom.

Figure 2.4 Validation errors displayed on the order form

Now your Taco Cloud controllers not only display and capture input, but they also validate that the information meets some basic validation rules. Let's step back and reconsider the `HomeController` from chapter 1, looking at an alternative implementation.

2.4 Working with view controllers

Thus far, you've written three controllers for the Taco Cloud application. Although each controller serves a distinct purpose in the functionality of the application, they all pretty much follow the same programming model:

- They're all annotated with `@Controller` to indicate that they're controller classes that should be automatically discovered by Spring component scanning and instantiated as beans in the Spring application context.
- All but `HomeController` are annotated with `@RequestMapping` at the class level to define a baseline request pattern that the controller will handle.
- They all have one or more methods that are annotated with `@GetMapping` or `@PostMapping` to provide specifics on which methods should handle which kinds of requests.

Most of the controllers you'll write will follow that pattern. But when a controller is simple enough that it doesn't populate a model or process input—as is the case with your `HomeController`—there's another way that you can define the controller. Have a look at the next listing to see how you can declare a view controller—a controller that does nothing but forward the request to a view.

Listing 2.15 Declaring a view controller

```
package tacos.web;

import org.springframework.context.annotation.Configuration;
import
    org.springframework.web.servlet.config.annotation.ViewControllerRegistry
    ;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```

The most significant thing to notice about `@WebConfig` is that it implements the `WebMvcConfigurer` interface. `WebMvcConfigurer` defines several methods for configuring Spring MVC. Even though it's an interface, it provides default implementations of all

the methods, so you only need to override the methods you need. In this case, you override `addViewControllers()`.

The `addViewControllers()` method is given a `ViewControllerRegistry` that you can use to register one or more view controllers. Here, you call `addViewController()` on the registry, passing in `"/"`, which is the path for which your view controller will handle GET requests. That method returns a `ViewControllerRegistration` object, on which you immediately call `setViewName()` to specify `home` as the view that a request for `"/"` should be forwarded to.

And just like that, you've been able to replace `HomeController` with a few lines in a configuration class. You can now delete `HomeController`, and the application should still behave as it did before. The only other change required is to revisit `HomeControllerTest` from chapter 1, removing the reference to `HomeController` from the `@WebMvcTest` annotation, so that the test class will compile without errors.

Here, you've created a new `WebConfig` configuration class to house the view controller declaration. But any configuration class can implement `WebMvcConfigurer` and override the `addViewController` method. For instance, you could have added the same view controller declaration to the bootstrap `TacoCloudApplication` class like this:

```
@SpringBootApplication
public class TacoCloudApplication implements WebMvcConfigurer {

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

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```

By extending an existing configuration class, you can avoid creating a new configuration class, keeping your project artifact count down. But I tend to prefer creating a new configuration class for each kind of configuration (web, data, security, and so on), keeping the application bootstrap configuration clean and simple.

Speaking of view controllers, and more generically the views that controllers forward requests to, so far you've been using Thymeleaf for all of your views. I like Thymeleaf a lot, but maybe you prefer a different template model for your application views. Let's have a look at Spring's many supported view options.

2.5 Choosing a view template library

For the most part, your choice of a view template library is a matter of personal taste. Spring is very flexible and supports many common templating options. With only a

few small exceptions, the template library you choose will itself have no idea that it's even working with Spring.³

Table 2.2 catalogs the template options supported by Spring Boot autoconfiguration.

Table 2.2 Supported template options

Template	Spring Boot starter dependency
FreeMarker	<code>spring-boot-starter-freemarker</code>
Groovy Templates	<code>spring-boot-starter-groovy-templates</code>
JavaServer Pages (JSP)	None (provided by Tomcat or Jetty)
Mustache	<code>spring-boot-starter-mustache</code>
Thymeleaf	<code>spring-boot-starter-thymeleaf</code>

Generally speaking, you select the view template library you want, add it as a dependency in your build, and start writing templates in the `/templates` directory (under the `src/main/resources` directory in a Maven- or Gradle-built project). Spring Boot will detect your chosen template library and automatically configure the components required for it to serve views for your Spring MVC controllers.

You've already done this with Thymeleaf for the Taco Cloud application. In chapter 1, you selected the Thymeleaf check box when initializing the project. This resulted in Spring Boot's Thymeleaf starter being included in the `pom.xml` file. When the application starts up, Spring Boot autoconfiguration detects the presence of Thymeleaf and automatically configures the Thymeleaf beans for you. All you had to do was start writing templates in `/templates`.

If you'd rather use a different template library, you simply select it at project initialization or edit your existing project build to include the newly chosen template library.

For example, let's say you wanted to use Mustache instead of Thymeleaf. No problem. Just visit the project `pom.xml` file and replace this,

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

with this:

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

³ One such exception is Thymeleaf's Spring Security dialect, which we'll talk about in chapter 4.

Of course, you'd need to make sure that you write all the templates with Mustache syntax instead of Thymeleaf tags. The specifics of working with Mustache (or any of the template language choices) is well outside of the scope of this book, but to give you an idea of what to expect, here's a snippet from a Mustache template that will render one of the ingredient groups in the taco design form:

```
<h3>Designate your wrap:</h3>
{{#wrap}}
<div>
  <input name="ingredients" type="checkbox" value="{{id}}"/>
  <span>{{name}}</span><br/>
</div>
{{/wrap}}
```

This is the Mustache equivalent of the Thymeleaf snippet in section 2.1.3. The {{#wrap}} block (which concludes with {{/wrap}}) iterates through a collection in the request attribute whose key is `wrap` and renders the embedded HTML for each item. The {{id}} and {{name}} tags reference the `id` and `name` properties of the item (which should be an `Ingredient`).

You'll notice in table 2.2 that JSP doesn't require any special dependency in the build. That's because the servlet container itself (Tomcat by default) implements the JSP specification, thus requiring no further dependencies.

But there's a gotcha if you choose to use JSP. As it turns out, Java servlet containers—including embedded Tomcat and Jetty containers—usually look for JSPs somewhere under `/WEB-INF`. But if you're building your application as an executable JAR file, there's no way to satisfy that requirement. Therefore, JSP is only an option if you're building your application as a WAR file and deploying it in a traditional servlet container. If you're building an executable JAR file, you must choose Thymeleaf, FreeMarker, or one of the other options in table 2.2.

2.5.1 Caching templates

By default, templates are only parsed once, when they're first used, and the results of that parse are cached for subsequent use. This is a great feature for production, as it prevents redundant template parsing on each request and thus improves performance.

That feature is not so awesome at development time, however. Let's say you fire up your application and hit the taco design page and decide to make a few changes to it. When you refresh your web browser, you'll still be shown the original version. The only way you can see your changes is to restart the application, which is quite inconvenient.

Fortunately, there's a way to disable caching. All you need to do is set a template-appropriate caching property to `false`. Table 2.3 lists the caching properties for each of the supported template libraries.

Table 2.3 Properties to enable/disable template caching

Template	Cache enable property
FreeMarker	<code>spring.freemarker.cache</code>
Groovy Templates	<code>spring.groovy.template.cache</code>
Mustache	<code>spring.mustache.cache</code>
Thymeleaf	<code>spring.thymeleaf.cache</code>

By default, all of these properties are set to `true` to enable caching. You can disable caching for your chosen template engine by setting its cache property to `false`. For example, to disable Thymeleaf caching, add the following line in `application.properties`:

```
spring.thymeleaf.cache=false
```

The only catch is that you'll want to be sure to remove this line (or set it to `true`) before you deploy your application to production. One option is to set the property in a profile. (We'll talk about profiles in chapter 5.)

A much simpler option is to use Spring Boot's DevTools, as we opted to do in chapter 1. Among the many helpful bits of development-time help offered by DevTools, it will disable caching for all template libraries but will disable itself (and thus reenable template caching) when your application is deployed.

Summary

- Spring offers a powerful web framework called Spring MVC that can be used to develop the web frontend for a Spring application.
- Spring MVC is annotation-based, enabling the declaration of request-handling methods with annotations such as `@RequestMapping`, `@GetMapping`, and `@PostMapping`.
- Most request-handling methods conclude by returning the logical name of a view, such as a Thymeleaf template, to which the request (along with any model data) is forwarded.
- Spring MVC supports validation through the Java Bean Validation API and implementations of the Validation API such as Hibernate Validator.
- View controllers can be used to handle HTTP GET requests for which no model data or processing is required.
- In addition to Thymeleaf, Spring supports a variety of view options, including FreeMarker, Groovy Templates, and Mustache.

Working with data



This chapter covers

- Using Spring's `JdbcTemplate`
- Inserting data with `SimpleJdbcInsert`
- Declaring JPA repositories with Spring Data

Most applications offer more than just a pretty face. Although the user interface may provide interaction with an application, it's the data it presents and stores that separates applications from static websites.

In the Taco Cloud application, you need to be able to maintain information about ingredients, tacos, and orders. Without a database to store this information, the application wouldn't be able to progress much further than what you developed in chapter 2.

In this chapter, you're going to add data persistence to the Taco Cloud application. You'll start by using Spring support for JDBC (Java Database Connectivity) to eliminate boilerplate code. Then you'll rework the data repositories to work with the JPA (Java Persistence API), eliminating even more code.

3.1 Reading and writing data with JDBC

For decades, relational databases and SQL have enjoyed their position as the leading choice for data persistence. Even though many alternative database types have emerged in recent years, the relational database is still a top choice for a general-purpose data store and will not likely be usurped from its position any time soon.

When it comes to working with relational data, Java developers have several options. The two most common choices are JDBC and the JPA. Spring supports both of these with abstractions, making working with either JDBC or JPA easier than it would be without Spring. In this section, we'll focus on how Spring supports JDBC, and then we'll look at Spring support for JPA in section 3.2.

Spring JDBC support is rooted in the `JdbcTemplate` class. `JdbcTemplate` provides a means by which developers can perform SQL operations against a relational database without all the ceremony and boilerplate typically required when working with JDBC.

To gain an appreciation of what `JdbcTemplate` does, let's start by looking at an example of how to perform a simple query in Java without `JdbcTemplate`.

Listing 3.1 Querying a database without `JdbcTemplate`

```
@Override
public Ingredient findOne(String id) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
        connection = dataSource.getConnection();
        statement = connection.prepareStatement(
            "select id, name, type from Ingredient");
        statement.setString(1, id);
        resultSet = statement.executeQuery();
        Ingredient ingredient = null;
        if(resultSet.next()) {
            ingredient = new Ingredient(
                resultSet.getString("id"),
                resultSet.getString("name"),
                Ingredient.Type.valueOf(resultSet.getString("type")));
        }
        return ingredient;
    } catch (SQLException e) {
        // ??? What should be done here ???
    } finally {
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {}
        }
        if (statement != null) {
            try {
                statement.close();
            } catch (SQLException e) {}
        }
    }
}
```

```

        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {}
        }
    }
    return null;
}

```

I assure you that somewhere in listing 3.1 there are a couple of lines that query the database for ingredients. But I'll bet you had a hard time spotting that query needle in the JDBC haystack. It's surrounded by code that creates a connection, creates a statement, and cleans up by closing the connection, statement, and result set.

To make matters worse, any number of things could go wrong when creating the connection or the statement, or when performing the query. This requires that you catch a `SQLException`, which may or may not be helpful in figuring out what went wrong or how to address the problem.

`SQLException` is a checked exception, which requires handling in a catch block. But the most common problems, such as failure to create a connection to the database or a mistyped query, can't possibly be addressed in a catch block and are likely to be rethrown for handling upstream. In contrast, consider the methods that use `JdbcTemplate`.

Listing 3.2 Querying a database with `JdbcTemplate`

```

private JdbcTemplate jdbct;

@Override
public Ingredient findOne(String id) {
    return jdbct.queryForObject(
        "select id, name, type from Ingredient where id=?",
        this::mapRowToIngredient, id);
}

private Ingredient mapRowToIngredient(ResultSet rs, int rowNum)
    throws SQLException {
    return new Ingredient(
        rs.getString("id"),
        rs.getString("name"),
        Ingredient.Type.valueOf(rs.getString("type")));
}

```

The code in listing 3.2 is clearly much simpler than the raw JDBC example in listing 3.1; there aren't any statements or connections being created. And, after the method is finished, there isn't any cleanup of those objects. Finally, there isn't any handling of exceptions that can't properly be handled in a catch block. What's left is code that's focused solely on performing a query (the call to `JdbcTemplate`'s `queryForObject()` method) and mapping the results to an `Ingredient` object (in the `mapRowToIngredient()` method).

The code in listing 3.2 is a snippet of what you need to do to use `JdbcTemplate` to persist and read data in the Taco Cloud application. Let's take the next steps necessary to outfit the application with JDBC persistence. We'll start by making a few tweaks to the domain objects.

3.1.1 Adapting the domain for persistence

When persisting objects to a database, it's generally a good idea to have one field that uniquely identifies the object. Your `Ingredient` class already has an `id` field, but you need to add `id` fields to both `Taco` and `Order`.

Moreover, it might be useful to know when a `Taco` is created and when an `Order` is placed. You'll also need to add a field to each object to capture the date and time that the objects are saved. The following listing shows the new `id` and `createdAt` fields needed in the `Taco` class.

Listing 3.3 Adding ID and timestamp fields to the Taco class

```
@Data  
public class Taco {  
  
    private Long id;  
  
    private Date createdAt;  
  
    ...  
}
```

Because you use Lombok to automatically generate accessor methods at runtime, there's no need to do anything more than declare the `id` and `createdAt` properties. They'll have appropriate getter and setter methods as needed at runtime. Similar changes are required in the `Order` class, as shown here:

```
@Data  
public class Order {  
  
    private Long id;  
  
    private Date placedAt;  
  
    ...  
}
```

Again, Lombok automatically generates the accessor methods, so these are the only changes required in `Order`. (If for some reason you choose not to use Lombok, you'll need to write these methods yourself.)

Your domain classes are now ready for persistence. Let's see how to use `JdbcTemplate` to read and write them to a database.

3.1.2 Working with `JdbcTemplate`

Before you can start using `JdbcTemplate`, you need to add it to your project classpath. This can easily be accomplished by adding Spring Boot’s JDBC starter dependency to the build:

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

You’re also going to need a database where your data will be stored. For development purposes, an embedded database will be just fine. I favor the H2 embedded database, so I’ve added the following dependency to the build:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

Later, you’ll see how to configure the application to use an external database. But for now, let’s move on to writing a repository that fetches and saves `Ingredient` data.

DEFINING JDBC REPOSITORIES

Your `Ingredient` repository needs to perform these operations:

- Query for all ingredients into a collection of `Ingredient` objects
- Query for a single `Ingredient` by its id
- Save an `Ingredient` object

The following `IngredientRepository` interface defines those three operations as method declarations:

```
package tacos.data;

import tacos.Ingredient;

public interface IngredientRepository {
    Iterable<Ingredient> findAll();
    Ingredient findOne(String id);
    Ingredient save(Ingredient ingredient);
}
```

Although the interface captures the essence of what you need an ingredient repository to do, you’ll still need to write an implementation of `IngredientRepository` that uses `JdbcTemplate` to query the database. The code shown next is the first step in writing that implementation.

Listing 3.4 Beginning an ingredient repository with JdbcTemplate

```
package tacos.data;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import tacos.Ingredient;

@Repository
public class JdbcIngredientRepository
    implements IngredientRepository {

    private JdbcTemplate jdbc;

    @Autowired
    public JdbcIngredientRepository(JdbcTemplate jdbc) {
        this.jdbc = jdbc;
    }

    ...

}
```

As you can see, `JdbcIngredientRepository` is annotated with `@Repository`. This annotation is one of a handful of stereotype annotations that Spring defines, including `@Controller` and `@Component`. By annotating `JdbcIngredientRepository` with `@Repository`, you declare that it should be automatically discovered by Spring component scanning and instantiated as a bean in the Spring application context.

When Spring creates the `JdbcIngredientRepository` bean, it injects it with `JdbcTemplate` via the `@Autowired` annotated construction. The constructor assigns `JdbcTemplate` to an instance variable that will be used in other methods to query and insert into the database. Speaking of those other methods, let's take a look at the implementations of `findAll()` and `findById()`.

Listing 3.5 Querying the database with JdbcTemplate

```
@Override
public Iterable<Ingredient> findAll() {
    return jdbc.query("select id, name, type from Ingredient",
        this::mapRowToIngredient);
}

@Override
public Ingredient findOne(String id) {
    return jdbc.queryForObject(
        "select id, name, type from Ingredient where id=?",
        this::mapRowToIngredient, id);
}
```

```
private Ingredient mapRowToIngredient(ResultSet rs, int rowNum)
    throws SQLException {
    return new Ingredient(
        rs.getString("id"),
        rs.getString("name"),
        Ingredient.Type.valueOf(rs.getString("type")));
}
```

Both `findAll()` and `findById()` use `JdbcTemplate` in a similar way. The `findAll()` method, expecting to return a collection of objects, uses `JdbcTemplate`'s `query()` method. The `query()` method accepts the SQL for the query as well as an implementation of Spring's `RowMapper` for the purpose of mapping each row in the result set to an object. `findAll()` also accepts as its final argument(s) a list of any parameters required in the query. But, in this case, there aren't any required parameters.

The `findById()` method only expects to return a single `Ingredient` object, so it uses the `queryForObject()` method of `JdbcTemplate` instead of `query()`. `queryForObject()` works much like `query()` except that it returns a single object instead of a `List` of objects. In this case, it's given the query to perform, a `RowMapper`, and the `id` of `Ingredient` to fetch, which is used in place of the `?` in the query.

As shown in listing 3.5, the `RowMapper` parameter for both `findAll()` and `findById()` is given as a method reference to the `mapRowToIngredient()` method. Java 8's method references and lambdas are convenient when working with `JdbcTemplate` as an alternative to an explicit `RowMapper` implementation. But if for some reason you want or need an explicit `RowMapper`, then the following implementation of `findAll()` shows how to do that:

```
@Override
public Ingredient findOne(String id) {
    return jdbc.queryForObject(
        "select id, name, type from Ingredient where id=?",
        new RowMapper<Ingredient>() {
            public Ingredient mapRow(ResultSet rs, int rowNum)
                throws SQLException {
                return new Ingredient(
                    rs.getString("id"),
                    rs.getString("name"),
                    Ingredient.Type.valueOf(rs.getString("type")));
            }
        }, id);
}
```

Reading data from a database is only part of the story. At some point, data must be written to the database so that it can be read. So let's see about implementing the `save()` method.

INSERTING A ROW

`JdbcTemplate`'s `update()` method can be used for any query that writes or updates data in the database. And, as shown in the following listing, it can be used to insert data into the database.

Listing 3.6 Inserting data with JdbcTemplate

```

@Override
public Ingredient save(Ingredient ingredient) {
    jdbc.update(
        "insert into Ingredient (id, name, type) values (?, ?, ?)",
        ingredient.getId(),
        ingredient.getName(),
        ingredient.getType().toString());
    return ingredient;
}

```

Because it isn't necessary to map `ResultSet` data to an object, the `update()` method is much simpler than `query()` or `queryForObject()`. It only requires a `String` containing the SQL to perform as well as values to assign to any query parameters. In this case, the query has three parameters, which correspond to the final three parameters of the `save()` method, providing the ingredient's ID, name, and type.

With `JdbcIngredientRepository` complete, you can now inject it into `DesignTacoController` and use it to provide a list of `Ingredient` objects instead of using hardcoded values (as you did in chapter 2). The changes to `DesignTacoController` are shown next.

Listing 3.7 Injecting and using a repository in the controller

```

@Controller
@RequestMapping("/design")
@SessionAttributes("order")
public class DesignTacoController {

    private final IngredientRepository ingredientRepo;

    @Autowired
    public DesignTacoController(IngredientRepository ingredientRepo) {
        this.ingredientRepo = ingredientRepo;
    }

    @GetMapping
    public String showDesignForm(Model model) {
        List<Ingredient> ingredients = new ArrayList<>();
        ingredientRepo.findAll().forEach(i -> ingredients.add(i));

        Type[] types = Ingredient.Type.values();
        for (Type type : types) {
            model.addAttribute(type.toString().toLowerCase(),
                filterByType(ingredients, type));
        }
        return "design";
    }

    ...
}

```

Notice that the second line of the `showDesignForm()` method now makes a call to the injected `IngredientRepository`'s `findAll()` method. The `findAll()` method fetches all the ingredients from the database before filtering them into distinct types in the model.

You're almost ready to fire up the application and try these changes out. But before you can start reading data from the `Ingredient` table referenced in the queries, you should probably create that table and populate it with some ingredient data.

3.1.3 Defining a schema and preloading data

Aside from the `Ingredient` table, you're also going to need some tables that hold order and design information. Figure 3.1 illustrates the tables you'll need, as well as the relationships between those tables.

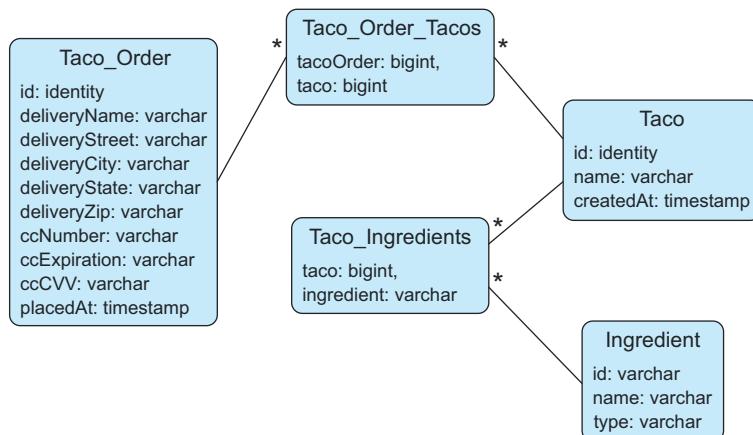


Figure 3.1 The tables for the Taco Cloud schema

The tables in figure 3.1 serve the following purposes:

- `Ingredient`—Holds ingredient information
- `Taco`—Holds essential information about a taco design
- `Taco_Ingredients`—Contains one or more rows for each row in `Taco`, mapping the taco to the ingredients for that taco
- `Taco_Order`—Holds essential order details
- `Taco_Order_Tacos`—Contains one or more rows for each row in `Taco_Order`, mapping the order to the tacos in the order

The next listing shows the SQL that creates the tables.

Listing 3.8 Defining the Taco Cloud schema

```
create table if not exists Ingredient (
    id varchar(4) not null,
    name varchar(25) not null,
    type varchar(10) not null
);

create table if not exists Taco (
    id identity,
    name varchar(50) not null,
    createdAt timestamp not null
);

create table if not exists Taco_Ingredients (
    taco bigint not null,
    ingredient varchar(4) not null
);

alter table Taco_Ingredients
    add foreign key (taco) references Taco(id);
alter table Taco_Ingredients
    add foreign key (ingredient) references Ingredient(id);

create table if not exists Taco_Order (
    id identity,
    deliveryName varchar(50) not null,
    deliveryStreet varchar(50) not null,
    deliveryCity varchar(50) not null,
    deliveryState varchar(2) not null,
    deliveryZip varchar(10) not null,
    ccNumber varchar(16) not null,
    ccExpiration varchar(5) not null,
    ccCVV varchar(3) not null,
    placedAt timestamp not null
);

create table if not exists Taco_Order_Tacos (
    tacoOrder bigint not null,
    taco bigint not null
);

alter table Taco_Order_Tacos
    add foreign key (tacoOrder) references Taco_Order(id);
alter table Taco_Order_Tacos
    add foreign key (taco) references Ingredient(id);
```

The big question is where to put this schema definition. As it turns out, Spring Boot answers that question.

If there's a file named `schema.sql` in the root of the application's classpath, then the SQL in that file will be executed against the database when the application starts. Therefore, you should place the contents of listing 3.8 in your project as a file named `schema.sql` in the `src/main/resources` folder.

You also need to preload the database with some ingredient data. Fortunately, Spring Boot will also execute a file named `data.sql` from the root of the classpath when the application starts. Therefore, you can load the database with ingredient data using the insert statements in the next listing, placed in `src/main/resources/data.sql`.

Listing 3.9 Preloading the database

```
delete from Taco_Order_Tacos;
delete from Taco_Ingredients;
delete from Taco;
delete from Taco_Order;

delete from Ingredient;
insert into Ingredient (id, name, type)
    values ('FLTO', 'Flour Tortilla', 'WRAP');
insert into Ingredient (id, name, type)
    values ('COTO', 'Corn Tortilla', 'WRAP');
insert into Ingredient (id, name, type)
    values ('GRBF', 'Ground Beef', 'PROTEIN');
insert into Ingredient (id, name, type)
    values ('CARN', 'Carnitas', 'PROTEIN');
insert into Ingredient (id, name, type)
    values ('TMTO', 'Diced Tomatoes', 'VEGGIES');
insert into Ingredient (id, name, type)
    values ('LETC', 'Lettuce', 'VEGGIES');
insert into Ingredient (id, name, type)
    values ('CHED', 'Cheddar', 'CHEESE');
insert into Ingredient (id, name, type)
    values ('JACK', 'Monterrey Jack', 'CHEESE');
insert into Ingredient (id, name, type)
    values ('SLSA', 'Salsa', 'SAUCE');
insert into Ingredient (id, name, type)
    values ('SRCR', 'Sour Cream', 'SAUCE');
```

Even though you've only developed a repository for ingredient data, you can fire up the Taco Cloud application at this point and visit the design page to see `JdbcIngredientRepository` in action. Go ahead ... give it a try. When you get back, you'll write the repositories for persisting Taco, Order, and data.

3.1.4 Inserting data

You've already had a glimpse into how to use `JdbcTemplate` to write data to the database. The `save()` method in `JdbcIngredientRepository` used the `update()` method of `JdbcTemplate` to save `Ingredient` objects to the database.

Although that was a good first example, it was perhaps a bit too simple. As you'll soon see, saving data can be more involved than what `JdbcIngredientRepository` needed. Two ways to save data with `JdbcTemplate` include the following:

- Directly, using the `update()` method
- Using the `SimpleJdbcInsert` wrapper class

Let's first see how to use the `update()` method when the persistence needs are more complex than what was required to save an `Ingredient`.

SAVING DATA WITH JDBCTEMPLATE

For now, the only thing that the `taco` and `order` repositories need to do is to save their respective objects. To save `Taco` objects, the `TacoRepository` declares a `save()` method like this:

```
package tacos.data;

import tacos.Taco;

public interface TacoRepository {
    Taco save(Taco design);
}
```

Similarly, `OrderRepository` also declares a `save()` method:

```
package tacos.data;

import tacos.Order;

public interface OrderRepository {
    Order save(Order order);
}
```

Seems simple enough, right? Not so quick. Saving a `taco` design requires that you also save the ingredients associated with that `taco` to the `Taco_Ingredients` table. Likewise, saving an `order` requires that you also save the `tacos` associated with the `order` to the `Taco_Order_Tacos` table. This makes saving `tacos` and `orders` a bit more challenging than what was required to save an `ingredient`.

To implement `TacoRepository`, you need a `save()` method that starts by saving the essential `taco` design details (for example, the name and time of creation), and then inserts one row into `Taco_Ingredients` for each ingredient in the `Taco` object. The following listing shows the complete `JdbcTacoRepository` class.

Listing 3.10 Implementing `TacoRepository` with `JdbcTemplate`

```
package tacos.data;

import java.sql.Timestamp;
import java.sql.Types;
import java.util.Arrays;
import java.util.Date;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
```

```
import org.springframework.jdbc.core.PreparedStatementCreatorFactory;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

import tacos.Ingredient;
import tacos.Taco;

@Repository
public class JdbcTacoRepository implements TacoRepository {

    private JdbcTemplate jdbc;

    public JdbcTacoRepository(JdbcTemplate jdbc) {
        this.jdbc = jdbc;
    }

    @Override
    public Taco save(Taco taco) {
        long tacoId = saveTacoInfo(taco);
        taco.setId(tacoId);
        for (Ingredient ingredient : taco.getIngredients()) {
            saveIngredientToTaco(ingredient, tacoId);
        }

        return taco;
    }

    private long saveTacoInfo(Taco taco) {
        taco.setCreatedAt(new Date());
        PreparedStatementCreator psc =
            new PreparedStatementCreatorFactory(
                "insert into Taco (name, createdAt) values (?, ?)",
                Types.VARCHAR, Types.TIMESTAMP
            ).newPreparedStatementCreator(
                Arrays.asList(
                    taco.getName(),
                    new Timestamp(taco.getCreatedAt().getTime())));
        KeyHolder keyHolder = new GeneratedKeyHolder();
        jdbc.update(psc, keyHolder);

        return keyHolder.getKey().longValue();
    }

    private void saveIngredientToTaco(
        Ingredient ingredient, long tacoId) {
        jdbc.update(
            "insert into Taco_Ingredients (taco, ingredient) " +
            "values (?, ?)",
            tacoId, ingredient.getId());
    }
}
```

As you can see, the `save()` method starts by calling the private `saveTacoInfo()` method, and then uses the taco ID returned from that method to call `saveIngredientToTaco()`, which saves each ingredient. The devil is in the details of `saveTacoInfo()`.

When you insert a row into Taco, you need to know the ID generated by the database so that you can reference it in each of the ingredients. The `update()` method, used when saving ingredient data, doesn't help you get at the generated ID, so you need a different `update()` method here.

The `update()` method you need accepts a `PreparedStatementCreator` and a `KeyHolder`. It's the `KeyHolder` that will provide the generated taco ID. But in order to use it, you must also create a `PreparedStatementCreator`.

As you can see from listing 3.10, creating a `PreparedStatementCreator` is non-trivial. Start by creating a `PreparedStatementCreatorFactory`, giving it the SQL you want to execute, as well as the types of each query parameter. Then call `newPreparedStatementCreator()` on that factory, passing in the values needed in the query parameters to produce the `PreparedStatementCreator`.

With a `PreparedStatementCreator` in hand, you can call `update()`, passing in `PreparedStatementCreator` and `KeyHolder` (in this case, a `GeneratedKeyHolder` instance). Once the `update()` is finished, you can return the taco ID by returning `keyHolder.getKey().longValue()`.

Back in `save()`, cycle through each `Ingredient` in `Taco`, calling `saveIngredientToTaco()`. The `saveIngredientToTaco()` method uses the simpler form of `update()` to save ingredient references to the `Taco_Ingredients` table.

All that's left to do with `TacoRepository` is to inject it into `DesignTacoController` and use it when saving tacos. The following listing shows the changes necessary for injecting the repository.

Listing 3.11 Injecting and using `TacoRepository`

```
@Controller
@RequestMapping("/design")
@SessionAttributes("order")
public class DesignTacoController {

    private final IngredientRepository ingredientRepo;
    private TacoRepository designRepo;

    @Autowired
    public DesignTacoController(
        IngredientRepository ingredientRepo,
        TacoRepository designRepo) {
        this.ingredientRepo = ingredientRepo;
        this.designRepo = designRepo;
    }

    ...
}
```

As you can see, the constructor takes both an `IngredientRepository` and a `TacoRepository`. It assigns both to instance variables so that they can be used in the `showDesignForm()` and `processDesign()` methods.

Speaking of the `processDesign()` method, its changes are a bit more extensive than the changes you made to `showDesignForm()`. The next listing shows the new `processDesign()` method.

Listing 3.12 Saving taco designs and linking them to orders

```

@Controller
@RequestMapping("/design")
@SessionAttributes("order")
public class DesignTacoController {

    @ModelAttribute(name = "order")
    public Order order() {
        return new Order();
    }

    @ModelAttribute(name = "taco")
    public Taco taco() {
        return new Taco();
    }

    @PostMapping
    public String processDesign(
        @Valid Taco design, Errors errors,
        @ModelAttribute Order order) {

        if (errors.hasErrors()) {
            return "design";
        }

        Taco saved = designRepo.save(design);
        order.addDesign(saved);

        return "redirect:/orders/current";
    }

    ...
}

```

The first thing you'll notice about the code in listing 3.12 is that `DesignTacoController` is now annotated with `@SessionAttributes("order")` and that it has a new `@ModelAttribute` annotated method, `order()`. As with the `taco()` method, the `@ModelAttribute` annotation on `order()` ensures that an `Order` object will be created in the model. But unlike the `Taco` object in the session, you need the order to be present across multiple requests so that you can create multiple tacos and add them to the order. The class-level `@SessionAttributes` annotation specifies any model

objects like the order attribute that should be kept in session and available across multiple requests.

The real processing of a taco design happens in the `processDesign()` method, which now accepts an `Order` object as a parameter, in addition to `Taco` and `Errors` objects. The `Order` parameter is annotated with `@ModelAttribute` to indicate that its value should come from the model and that Spring MVC shouldn't attempt to bind request parameters to it.

After checking for validation errors, `processDesign()` uses the injected `TacoRepository` to save the taco. It then adds the `Taco` object to the `Order` that's kept in the session.

In fact, the `Order` object remains in the session and isn't saved to the database until the user completes and submits the order form. At that point, `OrderController` needs to call out to an implementation of `OrderRepository` to save the order. Let's write that implementation.

INSERTING DATA WITH SIMPLEJDBCINSERT

You'll recall that saving a taco involved not only saving the taco's name and creation time to the `Taco` table, but also saving a reference to the ingredients associated with the taco to the `Taco_Ingredients` table. And you'll also recall that this required you to know the Taco's ID, which you obtained using `KeyHolder` and `PreparedStatementCreator`.

When it comes to saving orders, a similar circumstance exists. You must not only save the order data to the `Taco_Order` table, but also references to each taco in the order to the `Taco_Order_Tacos` table. But rather than use the cumbersome `PreparedStatementCreator`, allow me to introduce you to `SimpleJdbcInsert`, an object that wraps `JdbcTemplate` to make it easier to insert data into a table.

You'll start by creating `JdbcOrderRepository`, an implementation of `OrderRepository`. But before you write the `save()` method implementation, let's focus on the constructor, where you'll create a couple of instances of `SimpleJdbcInsert` for inserting values into the `Taco_Order` and `Taco_Order_Tacos` tables. The following listing shows `JdbcOrderRepository` (without the `save()` method).

Listing 3.13 Creating a SimpleJdbcInsert from a JdbcTemplate

```
package tacos.data;

import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
import org.springframework.stereotype.Repository;

import com.fasterxml.jackson.databind.ObjectMapper;
```

```

import tacos.Taco;
import tacos.Order;

@Repository
public class JdbcOrderRepository implements OrderRepository {

    private SimpleJdbcInsert orderInserter;
    private SimpleJdbcInsert orderTacoInserter;
    private ObjectMapper objectMapper;

    @Autowired
    public JdbcOrderRepository(JdbcTemplate jdbc) {
        this.orderInserter = new SimpleJdbcInsert(jdbc)
            .withTableName("Taco_Order")
            .usingGeneratedKeyColumns("id");

        this.orderTacoInserter = new SimpleJdbcInsert(jdbc)
            .withTableName("Taco_Order_Tacos");

        this.objectMapper = new ObjectMapper();
    }

    ...
}

}

```

Like `JdbcTacoRepository`, `JdbcOrderRepository` is injected with `JdbcTemplate` through its constructor. But instead of assigning `JdbcTemplate` directly to an instance variable, the constructor uses it to construct a couple of `SimpleJdbcInsert` instances.

The first instance, which is assigned to the `orderInserter` instance variable, is configured to work with the `Taco_Order` table and to assume that the `id` property will be provided or generated by the database. The second instance, assigned to `orderTacoInserter`, is configured to work with the `Taco_Order_Tacos` table but makes no claims about how any IDs will be generated in that table.

The constructor also creates an instance of Jackson's `ObjectMapper` and assigns it to an instance variable. Although Jackson is intended for JSON processing, you'll see in a moment how you'll repurpose it to help you as you save orders and their associated tacos.

Now let's take a look at how the `save()` method uses the `SimpleJdbcInsert` instances. The next listing shows the `save()` method, as well as a couple of private methods that `save()` delegates for the real work.

Listing 3.14 Using `SimpleJdbcInsert` to insert data

```

@Override
public Order save(Order order) {
    order.setPlacedAt(new Date());
    long orderId = saveOrderDetails(order);
    order.setId(orderId);
    List<Taco> tacos = order.getTacos();

```

```

        for (Taco taco : tacos) {
            saveTacoToOrder(taco, orderId);
        }

        return order;
    }

    private long saveOrderDetails(Order order) {
        @SuppressWarnings("unchecked")
        Map<String, Object> values =
            objectMapper.convertValue(order, Map.class);
        values.put("placedAt", order.getPlacedAt());

        long orderId =
            orderInserter
                .executeAndReturnKey(values)
                .longValue();
        return orderId;
    }

    private void saveTacoToOrder(Taco taco, long orderId) {
        Map<String, Object> values = new HashMap<>();
        values.put("tacoOrder", orderId);
        values.put("taco", taco.getId());
        orderTacoInserter.execute(values);
    }
}

```

The `save()` method doesn't actually save anything. It defines the flow for saving an `Order` and its associated `Taco` objects, and delegates the persistence work to `saveOrderDetails()` and `saveTacoToOrder()`.

`SimpleJdbcInsert` has a couple of useful methods for executing the insert: `execute()` and `executeAndReturnKey()`. Both accept a `Map<String, Object>`, where the map keys correspond to the column names in the table the data is inserted into. The map values are inserted into those columns.

It's easy to create such a `Map` by copying the values from `Order` into entries of the `Map`. But `Order` has several properties, and those properties all share the same name with the columns that they're going into. Because of that, in `saveOrderDetails()`, I've decided to use Jackson's `ObjectMapper` and its `convertValue()` method to convert an `Order` into a `Map`.¹ Once the `Map` is created, you'll set the `placedAt` entry to the value of the `Order` object's `placedAt` property. This is necessary because `ObjectMapper` would otherwise convert the `Date` property into a `long`, which is incompatible with the `placedAt` field in the `Taco_Order` table.

With a `Map` full of order data ready, you can now call `executeAndReturnKey()` on `orderInserter`. This saves the order information to the `Taco_Order` table and returns

¹ I'll admit that this is a hackish use of `ObjectMapper`, but you already have Jackson in the classpath; Spring Boot's web starter brings it in. Also, using `ObjectMapper` to map an object into a `Map` is much easier than copying each property from the object into the `Map`. Feel free to replace the use of `ObjectMapper` with any code you prefer that builds the `Map` you'll give to the inserter objects.

the database-generated ID as a Number object, which a call to `longValue()` converts to a long returned from the method.

The `saveTacoToOrder()` method is significantly simpler. Rather than use the `ObjectMapper` to convert an object to a Map, you create the Map and set the appropriate values. Once again, the map keys correspond to column names in the table. A simple call to the `orderTacoInserter`'s `execute()` method performs the insert.

Now you can inject `OrderRepository` into `OrderController` and start using it. The following listing shows the complete `OrderController`, including the changes to use an injected `OrderRepository`.

Listing 3.15 Using an OrderRepository in OrderController

```
package tacos.web;
import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

import tacos.Order;
import tacos.data.OrderRepository;

@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
public class OrderController {

    private OrderRepository orderRepo;

    public OrderController(OrderRepository orderRepo) {
        this.orderRepo = orderRepo;
    }

    @GetMapping("/current")
    public String orderForm() {
        return "orderForm";
    }

    @PostMapping
    public String processOrder(@Valid Order order, Errors errors,
            SessionStatus sessionStatus) {
        if (errors.hasErrors()) {
            return "orderForm";
        }

        orderRepo.save(order);
        sessionStatus.setComplete();
    }
}
```

```
        return "redirect:/";
    }
}
```

Aside from injecting `OrderRepository` into the controller, the only significant changes in `OrderController` are in the `processOrder()` method. Here, the `Order` object submitted in the form (which also happens to be the same `Order` object maintained in session) is saved via the `save()` method on the injected `OrderRepository`.

Once the order is saved, you don't need it hanging around in a session anymore. In fact, if you don't clean it out, the order remains in session, including its associated tacos, and the next order will start with whatever tacos the old order contained. Therefore, the `processOrder()` method asks for a `SessionStatus` parameter and calls its `setComplete()` method to reset the session.

All of the JDBC persistence code is in place. Now you can fire up the Taco Cloud application and try it out. Feel free to create as many tacos and as many orders as you'd like.

You might also find it helpful to dig around in the database. Because you're using H2 as your embedded database, and because you have Spring Boot DevTools in place, you should be able to point your browser to <http://localhost:8080/h2-console> to see the H2 Console. The default credentials should get you in, although you'll need to be sure that the JDBC URL field is set to `jdbc:h2:mem:testdb`. Once logged in, you should be able to issue any query you like against the tables in the Taco Cloud schema.

Spring's `JdbcTemplate`, along with `SimpleJdbcInsert`, makes working with relational databases significantly simpler than plain vanilla JDBC. But you may find that JPA makes it even easier. Let's rewind your work and see how to use Spring Data to make data persistence even easier.

3.2 Persisting data with Spring Data JPA

The Spring Data project is a rather large umbrella project comprised of several sub-projects, most of which are focused on data persistence with a variety of different database types. A few of the most popular Spring Data projects include these:

- *Spring Data JPA*—JPA persistence against a relational database
- *Spring Data MongoDB*—Persistence to a Mongo document database
- *Spring Data Neo4j*—Persistence to a Neo4j graph database
- *Spring Data Redis*—Persistence to a Redis key-value store
- *Spring Data Cassandra*—Persistence to a Cassandra database

One of the most interesting and useful features provided by Spring Data for all of these projects is the ability to automatically create repositories, based on a repository specification interface.

To see how Spring Data works, you’re going to start over, replacing the JDBC-based repositories from earlier in this chapter with repositories created by Spring Data JPA. But first, you need to add Spring Data JPA to the project build.

3.2.1 Adding Spring Data JPA to the project

Spring Data JPA is available to Spring Boot applications with the JPA starter. This starter dependency not only brings in Spring Data JPA, but also transitively includes Hibernate as the JPA implementation:

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

If you want to use a different JPA implementation, then you’ll need to, at least, exclude the Hibernate dependency and include the JPA library of your choice. For example, to use EclipseLink instead of Hibernate, you’ll need to alter the build as follows:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>hibernate-entitymanager</artifactId>
            <groupId>org.hibernate</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>2.5.2</version>
</dependency>
```

Note that there may be other changes required, depending on your choice of JPA implementation. Consult the documentation for your chosen JPA implementation for details. Now let’s revisit your domain objects and annotate them for JPA persistence.

3.2.2 Annotating the domain as entities

As you’ll soon see, Spring Data does some amazing things when it comes to creating repositories. But unfortunately, it doesn’t help much when it comes to annotating your domain objects with JPA mapping annotations. You’ll need to open up the `Ingredient`, `Taco`, and `Order` classes and throw in a few annotations. First up is the `Ingredient` class.

Listing 3.16 Annotating Ingredient for JPA persistence

```
package tacos;

import javax.persistence.Entity;
import javax.persistence.Id;

import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Entity
public class Ingredient {

    @Id
    private final String id;
    private final String name;
    private final Type type;

    public static enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }

}
```

In order to declare this as a JPA entity, `Ingredient` must be annotated with `@Entity`. And its `id` property must be annotated with `@Id` to designate it as the property that will uniquely identify the entity in the database.

In addition to the JPA-specific annotations, you'll also note that you've added a `@NoArgsConstructor` annotation at the class level. JPA requires that entities have a no-arguments constructor, so Lombok's `@NoArgsConstructor` does that for you. You don't want to be able to use it, though, so you make it private by setting the access attribute to `AccessLevel.PRIVATE`. And because there are final properties that must be set, you also set the `force` attribute to `true`, which results in the Lombok-generated constructor setting them to null.

You also add a `@RequiredArgsConstructor`. The `@Data` implicitly adds a required arguments constructor, but when a `@NoArgsConstructor` is used, that constructor gets removed. An explicit `@RequiredArgsConstructor` ensures that you'll still have a required arguments constructor in addition to the private no-arguments constructor.

Now let's move on to the `Taco` class and see how to annotate it as a JPA entity.

Listing 3.17 Annotating Taco as an entity

```
package tacos;
import java.util.Date;
import java.util.List;
```

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;
import javax.persistence.PrePersist;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import lombok.Data;

@Data
@Entity
public class Taco {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @NotNull
    @Size(min=5, message="Name must be at least 5 characters long")
    private String name;

    private Date createdAt;

    @ManyToMany(targetEntity=Ingredient.class)
    @Size(min=1, message="You must choose at least 1 ingredient")
    private List<Ingredient> ingredients;

    @PrePersist
    void createdAt() {
        this.createdAt = new Date();
    }
}
```

As with `Ingredient`, the `Taco` class is now annotated with `@Entity` and has its `id` property annotated with `@Id`. Because you're relying on the database to automatically generate the ID value, you also annotate the `id` property with `@GeneratedValue`, specifying a strategy of `AUTO`.

To declare the relationship between a `Taco` and its associated `Ingredient` list, you annotate `ingredients` with `@ManyToMany`. A `Taco` can have many `Ingredient` objects, and an `Ingredient` can be a part of many `Tacos`.

You'll also notice that there's a new method, `createdAt()`, which is annotated with `@PrePersist`. You'll use this to set the `createdAt` property to the current date and time before `Taco` is persisted. Finally, let's annotate the `Order` object as an entity. The next listing shows the new `Order` class.

Listing 3.18 Annotating Order as a JPA entity

```
package tacos;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.PrePersist;
import javax.persistence.Table;
import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.NotBlank;
import lombok.Data;

@Data
@Entity
@Table(name="Taco_Order")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    private Date placedAt;

    ...

    @ManyToOne(targetEntity=Taco.class)
    private List<Taco> tacos = new ArrayList<>();

    public void addDesign(Taco design) {
        this.tacos.add(design);
    }

    @PrePersist
    void placedAt() {
        this.placedAt = new Date();
    }
}
```

As you can see, the changes to `Order` closely mirror the changes to `Taco`. But there's one new annotation at the class level: `@Table`. This specifies that `Order` entities should be persisted to a table named `Taco_Order` in the database.

Although you could have used this annotation on any of the entities, it's necessary with `Order`. Without it, JPA would default to persisting the entities to a table named `Order`, but `order` is a reserved word in SQL and would cause problems. Now that the entities are properly annotated, it's time to write your repositories.

3.2.3 Declaring JPA repositories

In the JDBC versions of the repositories, you explicitly declared the methods you wanted the repository to provide. But with Spring Data, you can extend the `CrudRepository` interface instead. For example, here's the new `IngredientRepository` interface:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
    extends CrudRepository<Ingredient, String> {

}
```

`CrudRepository` declares about a dozen methods for CRUD (create, read, update, delete) operations. Notice that it's parameterized, with the first parameter being the entity type the repository is to persist, and the second parameter being the type of the entity ID property. For `IngredientRepository`, the parameters should be `Ingredient` and `String`.

You can similarly define the `TacoRepository` like this:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Taco;

public interface TacoRepository
    extends CrudRepository<Taco, Long> {

}
```

The only significant differences between `IngredientRepository` and `TacoRepository` are the parameters to `CrudRepository`. Here, they're set to `Taco` and `Long` to specify the `Taco` entity (and its ID type) as the unit of persistence for this repository interface. Finally, the same changes can be applied to `OrderRepository`:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Order;
```

```
public interface OrderRepository  
    extends CrudRepository<Order, Long> {  
  
}
```

And now you have your three repositories. You might be thinking that you need to write the implementations for all three, including the dozen methods for each implementation. But that's the good news about Spring Data JPA—there's no need to write an implementation! When the application starts, Spring Data JPA automatically generates an implementation on the fly. This means the repositories are ready to use from the get-go. Just inject them into the controllers like you did for the JDBC-based implementations, and you're done.

The methods provided by `CrudRepository` are great for general-purpose persistence of entities. But what if you have some requirements beyond basic persistence? Let's see how to customize the repositories to perform queries unique to your domain.

3.2.4 Customizing JPA repositories

Imagine that in addition to the basic CRUD operations provided by `CrudRepository`, you also need to fetch all the orders delivered to a given ZIP code. As it turns out, this can easily be addressed by adding the following method declaration to `OrderRepository`:

```
List<Order> findByDeliveryZip(String deliveryZip);
```

When generating the repository implementation, Spring Data examines any methods in the repository interface, parses the method name, and attempts to understand the method's purpose in the context of the persisted object (an `Order`, in this case). In essence, Spring Data defines a sort of miniature domain-specific language (DSL) where persistence details are expressed in repository method signatures.

Spring Data knows that this method is intended to find `Orders`, because you've parameterized `CrudRepository` with `Order`. The method name, `findByDeliveryZip()`, makes it clear that this method should find all `Order` entities by matching their `deliveryZip` property with the value passed in as a parameter to the method.

The `findByDeliveryZip()` method is simple enough, but Spring Data can handle even more-interesting method names as well. Repository methods are composed of a verb, an optional subject, the word *By*, and a predicate. In the case of `findByDeliveryZip()`, the verb is *find* and the predicate is *DeliveryZip*; the subject isn't specified and is implied to be an `Order`.

Let's consider another, more complex example. Suppose that you need to query for all orders delivered to a given ZIP code within a given date range. In that case, the following method, when added to `OrderRepository`, might prove useful:

```
List<Order> readOrdersByDeliveryZipAndPlacedAtBetween(  
    String deliveryZip, Date startDate, Date endDate);
```

Figure 3.2 illustrates how Spring Data parses and understands the `readOrdersByDeliveryZipAndPlacedAtBetween()` method when generating the repository implementation. As you can see, the verb in `readOrdersByDeliveryZipAndPlacedAtBetween()` is read. Spring Data also understands find, read, and get as synonymous for fetching one or more entities. Alternatively, you can also use count as the verb if you only want the method to return an int with the count of matching entities.

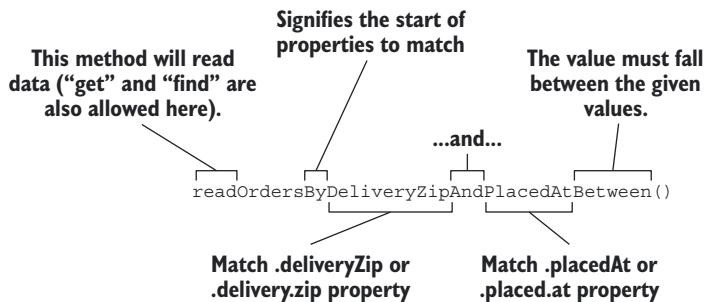


Figure 3.2 Spring Data parses repository method signatures to determine the query that should be performed.

Although the subject of the method is optional, here it says Orders. Spring Data ignores most words in a subject, so you could name the method `readPuppiesBy...` and it would still find Order entities, as that is the type that `CrudRepository` is parameterized with.

The predicate follows the word By in the method name and is the most interesting part of the method signature. In this case, the predicate refers to two Order properties: `deliveryZip` and `placedAt`. The `deliveryZip` property must be equal to the value passed into the first parameter of the method. The keyword Between indicates that the value of `deliveryZip` must fall between the values passed into the last two parameters of the method.

In addition to an implicit Equals operation and the Between operation, Spring Data method signatures can also include any of these operators:

- IsAfter, After, IsGreaterThan, GreaterThan
- IsGreaterThanOrEqualTo, GreaterThanEqual
- IsBefore, Before, IsLessThan, LessThan
- IsLessThanOrEqualTo, LessThanEqual
- IsBetween, Between
- IsNull, Null
- IsNotNotNull, NotNull
- IsIn, In
- IsNotIn, NotIn
- IsStartingWith, StartingWith, StartsWith

- `IsEndingWith`, `EndingWith`, `EndsWith`
- `IsContaining`, `Containing`, `Contains`
- `IsLike`, `Like`
- `IsNotLike`, `NotLike`
- `IsTrue`, `True`
- `IsFalse`, `False`
- `Is`, `Equals`
- `IsNot`, `Not`
- `IgnoringCase`, `IgnoresCase`

As alternatives for `IgnoringCase` and `IgnoresCase`, you can place either `AllIgnoringCase` or `AllIgnoresCase` on the method to ignore case for all String comparisons. For example, consider the following method:

```
List<Order> findByDeliveryToAndDeliveryCityAllIgnoresCase(  
    String deliveryTo, String deliveryCity);
```

Finally, you can also place `OrderBy` at the end of the method name to sort the results by a specified column. For example, to order by the `deliveryTo` property:

```
List<Order> findByDeliveryCityOrderByDeliveryTo(String city);
```

Although the naming convention can be useful for relatively simple queries, it doesn't take much imagination to see that method names could get out of hand for more-complex queries. In that case, feel free to name the method anything you want and annotate it with `@Query` to explicitly specify the query to be performed when the method is called, as this example shows:

```
@Query("Order o where o.deliveryCity='Seattle'")  
List<Order> readOrdersDeliveredInSeattle();
```

In this simple usage of `@Query`, you ask for all orders delivered in Seattle. But you can use `@Query` to perform virtually any query you can dream up, even when it's difficult or impossible to achieve the query by following the naming convention.

Summary

- Spring's `JdbcTemplate` greatly simplifies working with JDBC.
- `PreparedStatementCreator` and `KeyHolder` can be used together when you need to know the value of a database-generated ID.
- For easy execution of data inserts, use `SimpleJdbcInsert`.
- Spring Data JPA makes JPA persistence as easy as writing a repository interface.

Securing Spring



This chapter covers

- Autoconfiguring Spring Security
- Defining custom user storage
- Customizing the login page
- Securing against CSRF attacks
- Knowing your user

Have you ever noticed that most people in television sitcoms don't lock their doors? In the days of *Leave it to Beaver*, it wasn't so unusual for people to leave their doors unlocked. But it seems crazy that in a day when we're concerned with privacy and security, we see television characters enabling unhindered access to their apartments and homes.

Information is probably the most valuable item we now have; crooks are looking for ways to steal our data and identities by sneaking into unsecured applications. As software developers, we must take steps to protect the information that resides in our applications. Whether it's an email account protected with a username-password pair or a brokerage account protected with a trading PIN, security is a crucial aspect of most applications.

4.1 Enabling Spring Security

The very first step in securing your Spring application is to add the Spring Boot security starter dependency to your build. In the project's pom.xml file, add the following <dependency> entry:

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

If you're using Spring Tool Suite, this is even easier. Right-click on the pom.xml file and select Edit Starters from the Spring context menu. The Starter Dependencies dialog box will appear. Check the Security entry under the Core category, as shown in figure 4.1.

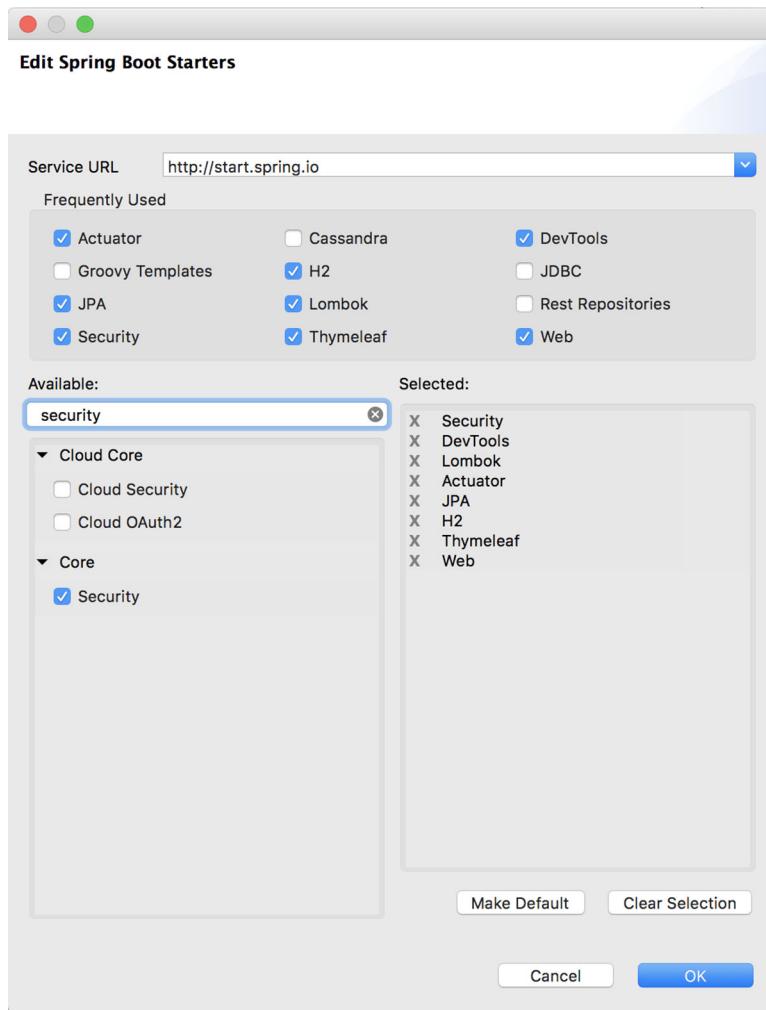


Figure 4.1 Adding the security starter with Spring Tool Suite

Believe it or not, that dependency is the only thing that's required to secure an application. When the application starts, autoconfiguration will detect that Spring Security is in the classpath and will set up some basic security configuration.

If you want to try it out, fire up the application and try to visit the homepage (or any page for that matter). You'll be prompted for authentication with an HTTP basic authentication dialog box. To get past it, you'll need to provide a username and password. The username is *user*. As for the password, it's randomly generated and written to the application log file. The log entry will look something like this:

```
Using default security password: 087cfcc6a-027d-44bc-95d7-cbb3a798a1ea
```

Assuming you enter the username and password correctly, you'll be granted access to the application.

It seems that securing Spring applications is pretty easy work. With the Taco Cloud application secured, I suppose I could end this chapter now and move on to the next topic. But before we get ahead of ourselves, let's consider what kind of security auto-configuration has provided.

By doing nothing more than adding the security starter to the project build, you get the following security features:

- All HTTP request paths require authentication.
- No specific roles or authorities are required.
- There's no login page.
- Authentication is prompted with HTTP basic authentication.
- There's only one user; the username is *user*.

This is a good start, but I think that the security needs of most applications (Taco Cloud included) will be quite different from these rudimentary security features.

You have more work to do if you're going to properly secure the Taco Cloud application. You'll need to *at least* configure Spring Security to do the following:

- Prompt for authentication with a login page, instead of an HTTP basic dialog box.
- Provide for multiple users, and enable a registration page so new Taco Cloud customers can sign up.
- Apply different security rules for different request paths. The homepage and registration pages, for example, shouldn't require authentication at all.

To meet your security needs for Taco Cloud, you'll have to write some explicit configuration, overriding what autoconfiguration has given you. You'll start by configuring a proper user store so that you can have more than one user.

4.2 Configuring Spring Security

Over the years there have been several ways of configuring Spring Security, including lengthy XML-based configuration. Fortunately, several recent versions of Spring Security have supported Java-based configuration, which is much easier to read and write.

Before this chapter is finished, you'll have configured all of your Taco Cloud security needs in Java-based Spring Security configuration. But to get started, you'll ease into it by writing the barebones configuration class shown in the following listing.

Listing 4.1 A barebones configuration class for Spring Security

```
package tacos.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web
    .configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web
    .configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

}
```

What does this barebones security configuration do for you? Well, not much, but it does move you a step closer to the security functionality you need. If you attempt to hit the Taco Cloud homepage again, you'll still be prompted to sign in. But instead of being prompted with an HTTP basic authentication dialog box, you'll be shown a login form like the one in figure 4.2.

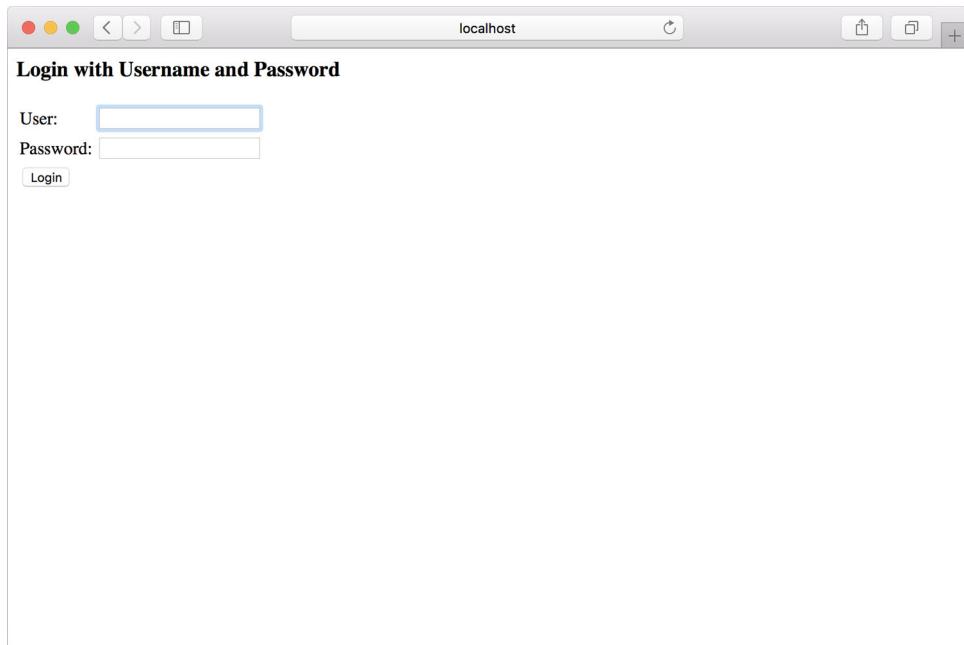


Figure 4.2 Spring Security gives you a plain login page for free.

TIP Going incognito: You may find it useful to set your browser to private or incognito mode when manually testing security. This will ensure that you have a fresh session each time you open a private/incognito window. You'll have to sign in to the application each time, but you can be assured that any changes you've made in security are applied, and that there aren't any remnants of an older session preventing you from seeing your changes.

This is a small improvement—prompting for login with a web page (even if it is rather plain in appearance) is always more user-friendly than an HTTP basic dialog box. You'll customize the login page in section 4.3.2. The current task at hand, however, is to configure a user store that can handle more than one user.

As it turns out, Spring Security offers several options for configuring a user store, including these:

- An in-memory user store
- A JDBC-based user store
- An LDAP-backed user store
- A custom user details service

No matter which user store you choose, you can configure it by overriding a `configure()` method defined in the `WebSecurityConfigurerAdapter` configuration base class. To get started, you'll add the following method override to the `SecurityConfig` class:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
    ...
}
```

Now you just need to replace those ellipses with code that uses the given `AuthenticationManagerBuilder` to specify how users will be looked up during authentication. First up, you'll try the in-memory user store.

4.2.1 In-memory user store

One place where user information can be kept is in memory. Suppose you have only a handful of users, none of which are likely to change. In that case, it may be simple enough to define those users as part of the security configuration.

For example, the next listing shows how to configure two users, "buzz" and "woody", in an in-memory user store.

Listing 4.2 Defining users in an in-memory user store

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
```

```
auth
    .inMemoryAuthentication()
        .withUser("buzz")
            .password("infinity")
            .authorities("ROLE_USER")
        .and()
        .withUser("woody")
            .password("bullseye")
            .authorities("ROLE_USER");
}
```

As you can see, `AuthenticationManagerBuilder` employs a builder-style API to configure authentication details. In this case, a call to the `inMemoryAuthentication()` method gives you an opportunity to specify user information directly in the security configuration itself.

Each call to `withUser()` starts the configuration for a user. The value given to `withUser()` is the username, whereas the password and granted authorities are specified with the `password()` and `authorities()` methods. As shown in listing 4.2, both users are granted `ROLE_USER` authority. User `buzz` is configured to have *infinity* as their password. Likewise, `woody`'s password is *bullseye*.

The in-memory user store is convenient for testing purposes or for very simple applications, but it doesn't allow for easy editing of users. If you need to add, remove, or change a user, you'll have to make the necessary changes and then rebuild and redeploy the application.

For the Taco Cloud application, you want customers to be able to register with the application and manage their own user accounts. That doesn't fit with the limitations of the in-memory user store, so let's take a look at another option that allows for a database-backed user store.

4.2.2 JDBC-based user store

User information is often maintained in a relational database, and a JDBC-based user store seems appropriate. The following listing shows how to configure Spring Security to authenticate against user information kept in a relational database with JDBC.

Listing 4.3 Authenticating against a JDBC-based user store

```
@Autowired
DataSource dataSource;

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

    auth
        .jdbcAuthentication()
        .dataSource(dataSource);
}
```

This implementation of `configure()` calls `jdbcAuthentication()` on the given `AuthenticationManagerBuilder`. From there, you must also set the `DataSource` so that it knows how to access the database. The `DataSource` used here is provided by the magic of autowiring.

OVERRIDING THE DEFAULT USER QUERIES

Although this minimal configuration will work, it makes some assumptions about your database schema. It expects that certain tables exist where user data will be kept. More specifically, the following snippet of code from Spring Security's internals shows the SQL queries that will be performed when looking up user details:

```
public static final String DEF_USERS_BY_USERNAME_QUERY =
    "select username,password(enabled " +
    "from users " +
    "where username = ?";
public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY =
    "select username,authority " +
    "from authorities " +
    "where username = ?";
public static final String DEF_GROUP_AUTHORITIES_BY_USERNAME_QUERY =
    "select g.id, g.group_name, ga.authority " +
    "from groups g, group_members gm, groupAuthorities ga " +
    "where gm.username = ? " +
    "and g.id = ga.group_id " +
    "and g.id = gm.group_id";
```

The first query retrieves a user's username, password, and whether or not they're enabled. This information is used to authenticate the user. The next query looks up the user's granted authorities for authorization purposes, and the final query looks up authorities granted to a user as a member of a group.

If you're OK with defining and populating tables in your database that satisfy those queries, there's not much else for you to do. But chances are your database doesn't look anything like this, and you'll want more control over the queries. In that case, you can configure your own queries.

Listing 4.4 Customizing user detail queries

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth
        .jdbcAuthentication()
        .dataSource(dataSource)
        .usersByUsernameQuery(
            "select username, password, enabled from Users " +
            "where username=?")
        .authoritiesByUsernameQuery(
            "select username, authority from UserAuthorities " +
            "where username=?");
}
```

In this case, you only override the authentication and basic authorization queries. But you can also override the group authorities query by calling `groupAuthoritiesByUsername()` with a custom query.

When replacing the default SQL queries with those of your own design, it's important to adhere to the basic contract of the queries. All of them take the username as their only parameter. The authentication query selects the username, password, and enabled status. The authorities query selects zero or more rows containing the username and a granted authority. The group authorities query selects zero or more rows, each with a group ID, a group name, and an authority.

WORKING WITH ENCODED PASSWORDS

Focusing on the authentication query, you can see that user passwords are expected to be stored in the database. The only problem with this is that if the passwords are stored in plain text, they're subject to the prying eyes of a hacker. But if you encode the passwords in the database, authentication will fail because it won't match the plaintext password submitted by the user.

To remedy this problem, you need to specify a password encoder by calling the `passwordEncoder()` method:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth)  
throws Exception {  
  
    auth  
        .jdbcAuthentication()  
            .dataSource(dataSource)  
            .usersByUsernameQuery(  
                "select username, password, enabled from Users " +  
                "where username=?")  
            .authoritiesByUsernameQuery(  
                "select username, authority from UserAuthorities " +  
                "where username=?")  
        .passwordEncoder(new StandardPasswordEncoder("53cr3t"));  
  
}
```

The `passwordEncoder()` method accepts any implementation of Spring Security's `PasswordEncoder` interface. Spring Security's cryptography module includes several such implementations:

- `BCryptPasswordEncoder`—Applies bcrypt strong hashing encryption
- `NoOpPasswordEncoder`—Applies no encoding
- `Pbkdf2PasswordEncoder`—Applies PBKDF2 encryption
- `SCryptPasswordEncoder`—Applies scrypt hashing encryption
- `StandardPasswordEncoder`—Applies SHA-256 hashing encryption

The preceding code uses `StandardPasswordEncoder`. But you can choose any of the other implementations or even provide your own custom implementation if none of

the out-of-the-box implementations meet your needs. The `PasswordEncoder` interface is rather simple:

```
public interface PasswordEncoder {
    String encode(CharSequence rawPassword);
    boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

No matter which password encoder you use, it's important to understand that the password in the database is never decoded. Instead, the password that the user enters at login is encoded using the same algorithm, and it's then compared with the encoded password in the database. That comparison is performed in the `PasswordEncoder`'s `matches()` method.

Ultimately, you'll maintain Taco Cloud user data in a database. Rather than use `jdbcAuthentication()`, however, I've got another authentication option in mind. But before we go there, let's look at how you can configure Spring Security to rely on another common source of user data: a user store accessed with LDAP (Lightweight Directory Access Protocol).

4.2.3 LDAP-backed user store

To configure Spring Security for LDAP-based authentication, you can use the `ldapAuthentication()` method. This method is the LDAP analog to `jdbcAuthentication()`. The following `configure()` method shows a simple configuration for LDAP authentication:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth
        .ldapAuthentication()
        .userSearchFilter("(uid={0})")
        .groupSearchFilter("member={0}");
}
```

The `userSearchFilter()` and `groupSearchFilter()` methods are used to provide filters for the base LDAP queries, which are used to search for users and groups. By default, the base queries for both users and groups are empty, indicating that the search will be done from the root of the LDAP hierarchy. But you can change that by specifying a query base:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth
        .ldapAuthentication()
        .userSearchBase("ou=people")
        .userSearchFilter("(uid={0})")
        .groupSearchBase("ou=groups")
        .groupSearchFilter("member={0}");
}
```

The `userSearchBase()` method provides a base query for finding users. Likewise, the `groupSearchBase()` method specifies the base query for finding groups. Rather than search from the root, this example specifies that users be searched for where the organizational unit is `people`. Groups should be searched for where the organizational unit is `groups`.

CONFIGURING PASSWORD COMPARISON

The default strategy for authenticating against LDAP is to perform a bind operation, authenticating the user directly to the LDAP server. Another option is to perform a comparison operation. This involves sending the entered password to the LDAP directory and asking the server to compare the password against a user's password attribute. Because the comparison is done within the LDAP server, the actual password remains secret.

If you'd rather authenticate by doing a password comparison, you can declare so with the `passwordCompare()` method:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth)  
    throws Exception {  
    auth  
        .ldapAuthentication()  
        .userSearchBase("ou=people")  
        .userSearchFilter("(uid={0})")  
        .groupSearchBase("ou=groups")  
        .groupSearchFilter("member={0}")  
        .passwordCompare();  
}
```

By default, the password given in the login form will be compared with the value of the `userPassword` attribute in the user's LDAP entry. If the password is kept in a different attribute, you can specify the password attribute's name with `passwordAttribute()`:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth)  
    throws Exception {  
    auth  
        .ldapAuthentication()  
        .userSearchBase("ou=people")  
        .userSearchFilter("(uid={0})")  
        .groupSearchBase("ou=groups")  
        .groupSearchFilter("member={0}")  
        .passwordCompare()  
        .passwordEncoder(new BCryptPasswordEncoder())  
        .passwordAttribute("passcode");  
}
```

In this example, you specify that the `passcode` attribute is what should be compared with the given password. Moreover, you also specify a password encoder. It's nice that the actual password is kept secret on the server when doing server-side password comparison. But the attempted password is still passed across the wire to the LDAP server

and could be intercepted by a hacker. To prevent that, you can specify an encryption strategy by calling the `passwordEncoder()` method.

In the preceding example, passwords are encrypted using the `bcrypt` password hashing function. This assumes that the passwords are also encrypted using `bcrypt` in the LDAP server.

REFERRING TO A REMOTE LDAP SERVER

The one thing I've left out until now is where the LDAP server and data actually reside. You've been happily configuring Spring to authenticate against an LDAP server, but where is that server?

By default, Spring Security's LDAP authentication assumes that the LDAP server is listening on port 33389 on localhost. But if your LDAP server is on another machine, you can use the `contextSource()` method to configure the location:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth)  
    throws Exception {  
    auth  
        .ldapAuthentication()  
            .userSearchBase("ou=people")  
            .userSearchFilter("(uid={0})")  
            .groupSearchBase("ou=groups")  
            .groupSearchFilter("member={0}")  
            .passwordCompare()  
            .passwordEncoder(new BCryptPasswordEncoder())  
            .passwordAttribute("passcode")  
            .contextSource()  
                .url("ldap://tacocloud.com:389/dc=tacocloud,dc=com");  
}
```

The `contextSource()` method returns a `ContextSourceBuilder`, which, among other things, offers the `url()` method, which lets you specify the location of the LDAP server.

CONFIGURING AN EMBEDDED LDAP SERVER

If you don't happen to have an LDAP server lying around waiting to be authenticated against, Spring Security can provide an embedded LDAP server for you. Instead of setting the URL to a remote LDAP server, you can specify the root suffix for the embedded server via the `root()` method:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth)  
    throws Exception {  
    auth  
        .ldapAuthentication()  
            .userSearchBase("ou=people")  
            .userSearchFilter("(uid={0})")  
            .groupSearchBase("ou=groups")  
            .groupSearchFilter("member={0}")  
            .passwordCompare()  
            .passwordEncoder(new BCryptPasswordEncoder())  
            .passwordAttribute("passcode")
```

```
    .contextSource()
        .root("dc=tacocloud,dc=com");
}
```

When the LDAP server starts, it will attempt to load data from any LDIF files that it can find in the classpath. LDIF (LDAP Data Interchange Format) is a standard way of representing LDAP data in a plain text file. Each record is composed of one or more lines, each containing a name:value pair. Records are separated from each other by blank lines.

If you'd rather that Spring not rummage through your classpath looking for any LDIF files it can find, you can be more explicit about which LDIF file gets loaded by calling the `ldif()` method:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth
        .ldapAuthentication()
            .userSearchBase("ou=people")
            .userSearchFilter("(uid={0})")
            .groupSearchBase("ou=groups")
            .groupSearchFilter("member={0}")
            .passwordCompare()
            .passwordEncoder(new BCryptPasswordEncoder())
            .passwordAttribute("passcode")
            .contextSource()
                .root("dc=tacocloud,dc=com")
                .ldif("classpath:users.ldif");
}
```

Here, you specifically ask the LDAP server to load its content from the `users.ldif` file at the root of the classpath. In case you're curious, here's an LDIF file that you could use to load the embedded LDAP server with user data:

```
dn: ou=groups,dc=tacocloud,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups
dn: ou=people,dc=tacocloud,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people
dn: uid=buzz,ou=people,dc=tacocloud,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Buzz Lightyear
sn: Lightyear
uid: buzz
userPassword: password
dn: cn=tacocloud,ou=groups,dc=tacocloud,dc=com
```

```
objectclass: top
objectclass: groupOfNames
cn: tacocloud
member: uid=buzz,ou=people,dc=tacocloud,dc=com
```

Spring Security's built-in user stores are convenient and cover some common use cases. But the Taco Cloud application needs something a bit special. When the out-of-the-box user stores don't meet your needs, you'll need to create and configure a custom user details service.

4.2.4 Customizing user authentication

In the last chapter, you settled on using Spring Data JPA as your persistence option for all taco, ingredient, and order data. It would thus make sense to persist user data in the same way. If you do so, the data will ultimately reside in a relational database, so you could use JDBC-based authentication. But it'd be even better to leverage the Spring Data repository used to store users.

First things first, though. Let's create the domain object and repository interface that represents and persists user information.

DEFINING THE USER DOMAIN AND PERSISTENCE

When Taco Cloud customers register with the application, they'll need to provide more than just a username and password. They'll also give you their full name, address, and phone number. This information can be used for a variety of purposes, including pre-populating the order form (not to mention potential marketing opportunities).

To capture all of that information, you'll create a `User` class, as follows.

Listing 4.5 Defining a user entity

```
package tacos;
import java.util.Arrays;
import java.util.Collection;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.
    SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Entity
@Data
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@RequiredArgsConstructor
public class User implements UserDetails {
```

```
private static final long serialVersionUID = 1L;

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private Long id;

private final String username;
private final String password;
private final String fullname;
private final String street;
private final String city;
private final String state;
private final String zip;
private final String phoneNumber;

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"));
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}

}
```

You've no doubt noticed that the `User` class is a bit more involved than any of the other entities defined in chapter 3. In addition to defining a handful of properties, `User` also implements the `UserDetails` interface from Spring Security.

Implementations of `UserDetails` will provide some essential user information to the framework, such as what authorities are granted to the user and whether the user's account is enabled or not.

The `getAuthorities()` method should return a collection of authorities granted to the user. The various `is__Expired()` methods return a boolean to indicate whether or not the user's account is enabled or expired.

For your `User` entity, the `getAuthorities()` method simply returns a collection indicating that all users will have been granted `ROLE_USER` authority. And, at least for

now, Taco Cloud has no need to disable users, so all of the `is____Expired()` methods return true to indicate that the users are active.

With the `User` entity defined, you now can define the repository interface:

```
package tacos.data;
import org.springframework.data.repository.CrudRepository;
import tacos.User;

public interface UserRepository extends CrudRepository<User, Long> {
    User findByUsername(String username);
}
```

In addition to the CRUD operations provided by extending `CrudRepository`, `UserRepository` defines a `findByUsername()` method that you'll use in the user details service to look up a `User` by their username.

As you learned in chapter 3, Spring Data JPA will automatically generate the implementation of this interface at runtime. Therefore, you're now ready to write a custom user details service that uses this repository.

CREATING A USER-DETAILS SERVICE

Spring Security's `UserDetailsService` is a rather straightforward interface:

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException;
}
```

As you can see, implementations of this interface are given a user's username and are expected to either return a `UserDetails` object or throw a `UsernameNotFoundException` if the given username doesn't turn up any results.

Because your `User` class implements `UserDetails`, and because `UserRepository` provides a `findByUsername()` method, they're perfectly suitable for use in a custom `UserDetailsService` implementation. The following listing shows the user details service you'll use in the Taco Cloud application.

Listing 4.6 Defining a custom user details service

```
package tacos.security;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import tacos.User;
import tacos.data.UserRepository;
```

```
@Service
public class UserRepositoryUserDetailsService
    implements UserDetailsService {

    private UserRepository userRepo;

    @Autowired
    public UserRepositoryUserDetailsService(UserRepository userRepo) {
        this.userRepo = userRepo;
    }

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        User user = userRepo.findByUsername(username);
        if (user != null) {
            return user;
        }
        throw new UsernameNotFoundException(
            "User '" + username + "' not found");
    }
}
```

UserRepositoryUserDetailsService is injected with an instance of UserRepository through its constructor. Then, in its `loadByUsername()` method, it calls `findByUsername()` on the UserRepository to look up a User.

The `loadByUsername()` method has one simple rule: it must never return null. Therefore, if the call to `findByUsername()` returns null, `loadByUsername()` will throw a `UsernameNotFoundException`. Otherwise, the User that was found will be returned.

You'll notice that UserRepositoryUserDetailsService is annotated with `@Service`. This is another one of Spring's stereotype annotations that flag it for inclusion in Spring's component scanning, so there's no need to explicitly declare this class as a bean. Spring will automatically discover it and instantiate it as a bean.

You do, however, still need to configure your custom user details service with Spring Security. Therefore, you'll return to the `configure()` method once more:

```
@Autowired
private UserDetailsService userDetailsService;

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

    auth
        .userDetailsService(userDetailsService);
}
```

This time, you simply make a call to the `userDetailsService()` method, passing in the `UserDetailsService` instance that has been autowired into `SecurityConfig`.

As with JDBC-based authentication, you can (and should) also configure a password encoder so that the password can be encoded in the database. You'll do this by first declaring a bean of type `PasswordEncoder` and then injecting it into your user details service configuration by calling `passwordEncoder()`:

```
@Bean
public PasswordEncoder encoder() {
    return new StandardPasswordEncoder("53cr3t");
}

@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {

    auth
        .userDetailsService(userDetailsService)
        .passwordEncoder(encoder());
}
```

It's important that we discuss the last line in the `configure()` method. It would appear that you call the `encoder()` method and pass its return value to `passwordEncoder()`. In reality, however, because the `encoder()` method is annotated with `@Bean`, it will be used to declare a `PasswordEncoder` bean in the Spring application context. Any calls to `encoder()` will then be intercepted to return the bean instance from the application context.

Now that you have a custom user details service that reads user information via a JPA repository, you just need a way to get users into the database in the first place. You need to create a registration page for Taco Cloud patrons to register with the application.

REGISTERING USERS

Although Spring Security handles many aspects of security, it really isn't directly involved in the process of user registration, so you're going to rely on a little bit of Spring MVC to handle that task. The `RegistrationController` class in the following listing presents and processes registration forms.

Listing 4.7 A user registration controller

```
package tacos.security;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import tacos.data.UserRepository;

@Controller
@RequestMapping("/register")
public class RegistrationController {
```

```
private UserRepository userRepo;
private PasswordEncoder passwordEncoder;

public RegistrationController(
    UserRepository userRepo, PasswordEncoder passwordEncoder) {
    this.userRepo = userRepo;
    this.passwordEncoder = passwordEncoder;
}

@GetMapping
public String registerForm() {
    return "registration";
}

@PostMapping
public String processRegistration(RegistrationForm form) {
    userRepo.save(form.toUser(passwordEncoder));
    return "redirect:/login";
}
}
```

Like any typical Spring MVC controller, `RegistrationController` is annotated with `@Controller` to designate it as a controller and to mark it for component scanning. It's also annotated with `@RequestMapping` such that it will handle requests whose path is `/register`.

More specifically, a GET request for `/register` will be handled by the `registerForm()` method, which simply returns a logical view name of `registration`. The following listing shows a Thymeleaf template that defines the `registration` view.

Listing 4.8 A Thymeleaf registration form view

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Taco Cloud</title>
    </head>

    <body>
        <h1>Register</h1>
        

        <form method="POST" th:action="@{/register}" id="registerForm">

            <label for="username">Username: </label>
            <input type="text" name="username"/><br/>

            <label for="password">Password: </label>
            <input type="password" name="password"/><br/>

            <label for="confirm">Confirm password: </label>
            <input type="password" name="confirm"/><br/>
        
```

```
<label for="fullname">Full name: </label>
<input type="text" name="fullname"/><br/>

<label for="street">Street: </label>
<input type="text" name="street"/><br/>

<label for="city">City: </label>
<input type="text" name="city"/><br/>

<label for="state">State: </label>
<input type="text" name="state"/><br/>

<label for="zip">Zip: </label>
<input type="text" name="zip"/><br/>

<label for="phone">Phone: </label>
<input type="text" name="phone"/><br/>

<input type="submit" value="Register"/>
</form>

</body>
</html>
```

When the form is submitted, the HTTP POST request will be handled by the `processRegistration()` method. The `RegistrationForm` object given to `processRegistration()` is bound to the request data and is defined with the following class:

```
package tacos.security;
import org.springframework.security.crypto.password.PasswordEncoder;
import lombok.Data;
import tacos.User;

@Data
public class RegistrationForm {

    private String username;
    private String password;
    private String fullname;
    private String street;
    private String city;
    private String state;
    private String zip;
    private String phone;

    public User toUser(PasswordEncoder passwordEncoder) {
        return new User(
            username, passwordEncoder.encode(password),
            fullname, street, city, state, zip, phone);
    }
}
```

For the most part, `RegistrationForm` is just a basic Lombok-enabled class with a handful of properties. But the `toUser()` method uses those properties to create a new `User` object, which is what `processRegistration()` will save, using the injected `UserRepository`.

You've no doubt noticed that `RegistrationController` is injected with a `PasswordEncoder`. This is the exact same `PasswordEncoder` bean you declared before. When processing a form submission, `RegistrationController` passes it to the `toUser()` method, which uses it to encode the password before saving it to the database. In this way, the submitted password is written in an encoded form, and the user details service will be able to authenticate against that encoded password.

Now the Taco Cloud application has complete user registration and authentication support. But if you start it up at this point, you'll notice that you can't even get to the registration page without being prompted to log in. That's because, by default, all requests require authentication. Let's look at how web requests are intercepted and secured so you can fix this strange chicken-and-egg situation.

4.3 Securing web requests

The security requirements for Taco Cloud should require that a user be authenticated before designing tacos or placing orders. But the homepage, login page, and registration page should be available to unauthenticated users.

To configure these security rules, let me introduce you to `WebSecurityConfigurerAdapter`'s other `configure()` method:

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

This `configure()` method accepts an `HttpSecurity` object, which can be used to configure how security is handled at the web level. Among the many things you can configure with `HttpSecurity` are these:

- Requiring that certain security conditions be met before allowing a request to be served
- Configuring a custom login page
- Enabling users to log out of the application
- Configuring cross-site request forgery protection

Intercepting requests to ensure that the user has proper authority is one of the most common things you'll configure `HttpSecurity` to do. Let's ensure that your Taco Cloud customers meet those requirements.

4.3.1 Securing requests

You need to ensure that requests for /design and /orders are only available to authenticated users; all other requests should be permitted for all users. The following `configure()` implementation does exactly that:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/design", "/orders")
                .hasRole("ROLE_USER")
            .antMatchers("/*", "/**") .permitAll()
        ;
}
```

The call to `authorizeRequests()` returns an object (`ExpressionInterceptUrlRegistry`) on which you can specify URL paths and patterns and the security requirements for those paths. In this case, you specify two security rules:

- Requests for /design and /orders should be for users with a granted authority of `ROLE_USER`.
- All requests should be permitted to all users.

The order of these rules is important. Security rules declared first take precedence over those declared lower down. If you were to swap the order of those two security rules, all requests would have `permitAll()` applied to them; the rule for /design and /orders requests would have no effect.

The `hasRole()` and `permitAll()` methods are just a couple of the methods for declaring security requirements for request paths. Table 4.1 describes all the available methods.

Table 4.1 Configuration methods to define how a path is to be secured

Method	What it does
<code>access(String)</code>	Allows access if the given SpEL expression evaluates to <code>true</code>
<code>anonymous()</code>	Allows access to anonymous users
<code>authenticated()</code>	Allows access to authenticated users
<code>denyAll()</code>	Denies access unconditionally
<code>fullyAuthenticated()</code>	Allows access if the user is fully authenticated (not remembered)
<code>hasAnyAuthority(String...)</code>	Allows access if the user has any of the given authorities
<code>hasAnyRole(String...)</code>	Allows access if the user has any of the given roles
<code>hasAuthority(String)</code>	Allows access if the user has the given authority
<code>hasIpAddress(String)</code>	Allows access if the request comes from the given IP address

Table 4.1 Configuration methods to define how a path is to be secured (continued)

Method	What it does
hasRole(String)	Allows access if the user has the given role
not()	Negates the effect of any of the other access methods
permitAll()	Allows access unconditionally
rememberMe()	Allows access for users who are authenticated via remember-me

Most of the methods in table 4.1 provide essential security rules for request handling, but they're self-limiting, only enabling security rules as defined by those methods. Alternatively, you can use the `access()` method to provide a SpEL expression to declare richer security rules. Spring Security extends SpEL to include several security-specific values and functions, as listed in table 4.2.

Table 4.2 Spring Security extensions to the Spring Expression Language

Security expression	What it evaluates to
<code>authentication</code>	The user's authentication object
<code>denyAll</code>	Always evaluates to <code>false</code>
<code>hasAnyRole(list of roles)</code>	true if the user has any of the given roles
<code>hasRole(role)</code>	true if the user has the given role
<code>hasIpAddress(IP address)</code>	true if the request comes from the given IP address
<code>isAnonymous()</code>	true if the user is anonymous
<code>isAuthenticated()</code>	true if the user is authenticated
<code>isFullyAuthenticated()</code>	true if the user is fully authenticated (not authenticated with remember-me)
<code>isRememberMe()</code>	true if the user was authenticated via remember-me
<code>permitAll</code>	Always evaluates to <code>true</code>
<code>principal</code>	The user's principal object

As you can see, most of the security expression extensions in table 4.2 correspond to similar methods in table 4.1. In fact, using the `access()` method along with the `hasRole()` and `permitAll` expressions, you can rewrite `configure()` as follows.

Listing 4.9 Using Spring expressions to define authorization rules

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

```

    .antMatchers("/design", "/orders")
        .access("hasRole('ROLE_USER')")
    .antMatchers("/", "/**").access("permitAll")
;
}

```

This may not seem like a big deal at first. After all, these expressions only mirror what you already did with method calls. But expressions can be much more flexible. For instance, suppose that (for some crazy reason) you only wanted to allow users with ROLE_USER authority to create new tacos on Tuesdays (for example, on Taco Tuesday); you could rewrite the expression as shown in this modified version of `configure()`:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/design", "/orders")
            .access("hasRole('ROLE_USER') && " +
                    "T(java.util.Calendar).getInstance().get(" +
                    "T(java.util.Calendar).DAY_OF_WEEK) == " +
                    "T(java.util.Calendar).TUESDAY")
        .antMatchers("/", "/**").access("permitAll")
;
}

```

With SpEL-based security constraints, the possibilities are virtually endless. I'll bet that you're already dreaming up interesting security constraints based on SpEL.

The authorization needs for the Taco Cloud application are met by the simple use of `access()` and the SpEL expressions in listing 4.9. Now let's see about customizing the login page to fit the look of the Taco Cloud application.

4.3.2 Creating a custom login page

The default login page is much better than the clunky HTTP basic dialog box you started with, but it's still rather plain and doesn't quite fit into the look of the rest of the Taco Cloud application.

To replace the built-in login page, you first need to tell Spring Security what path your custom login page will be at. That can be done by calling `formLogin()` on the `HttpSecurity` object passed into `configure()`:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/design", "/orders")
            .access("hasRole('ROLE_USER')")
        .antMatchers("/", "/**").access("permitAll")

        .and()
        .formLogin()
}

```

```

    .loginPage("/login")
;
}
}

```

Notice that before you call `formLogin()`, you bridge this section of configuration and the previous section with a call to `and()`. The `and()` method signifies that you're finished with the authorization configuration and are ready to apply some additional HTTP configuration. You'll use `and()` several times as you begin new sections of configuration.

After the bridge, you call `formLogin()` to start configuring your custom login form. The call to `loginPage()` after that designates the path where your custom login page will be provided. When Spring Security determines that the user is unauthenticated and needs to log in, it will redirect them to this path.

Now you need to provide a controller that handles requests at that path. Because your login page will be fairly simple—nothing but a view—it's easy enough to declare it as a view controller in `WebConfig`. The following `addViewControllers()` method sets up the login page view controller alongside the view controller that maps `"/"` to the home controller:

```

@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("home");
    registry.addViewController("/login");
}
}

```

Finally, you need to define the login page view itself. Because you're using Thymeleaf as your template engine, the following Thymeleaf template should do fine:

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Taco Cloud</title>
</head>

<body>
    <h1>Login</h1>
    

    <div th:if="${error}">
        Unable to login. Check your username and password.
    </div>

    <p>New here? Click
        <a th:href="@{/register}">here</a> to register.</p>

    <!-- tag::thAction[] -->
    <form method="POST" th:action="@{/login}" id="loginForm">
    <!-- end::thAction[] -->
        <label for="username">Username: </label>
        <input type="text" name="username" id="username" /><br/>

```

```

<label for="password">Password: </label>
<input type="password" name="password" id="password" /><br/>

<input type="submit" value="Login"/>
</form>
</body>
</html>

```

The key things to note about this login page are the path it posts to and the names of the username and password fields. By default, Spring Security listens for login requests at /login and expects that the username and password fields be named username and password. This is configurable, however. For example, the following configuration customizes the path and field names:

```

.and()
.formLogin()
.loginPage("/login")
.loginProcessingUrl("/authenticate")
.usernameParameter("user")
.passwordParameter("pwd")

```

Here, you specify that Spring Security should listen for requests to /authenticate to handle login submissions. Also, the username and password fields should now be named user and pwd.

By default, a successful login will take the user directly to the page that they were navigating to when Spring Security determined that they needed to log in. If the user were to directly navigate to the login page, a successful login would take them to the root path (for example, the homepage). But you can change that by specifying a default success page:

```

.and()
.formLogin()
.loginPage("/login")
.defaultSuccessUrl("/design")

```

As configured here, if the user were to successfully log in after directly going to the login page, they would be directed to the /design page.

Optionally, you can force the user to the design page after login, even if they were navigating elsewhere prior to logging in, by passing true as a second parameter to defaultSuccessUrl:

```

.and()
.formLogin()
.loginPage("/login")
.defaultSuccessUrl("/design", true)

```

Now that you've dealt with your custom login page, let's flip to the other side of the authentication coin and see how you can enable a user to log out.

4.3.3 Logging out

Just as important as logging into an application is logging out. To enable logout, you simply need to call `logout` on the `HttpSecurity` object:

```
.and()  
    .logout()  
        .logoutSuccessUrl("/")
```

This sets up a security filter that intercepts POST requests to `/logout`. Therefore, to provide logout capability, you just need to add a logout form and button to the views in your application:

```
<form method="POST" th:action="@{/logout}">  
    <input type="submit" value="Logout"/>  
</form>
```

When the user clicks the button, their session will be cleared, and they will be logged out of the application. By default, they'll be redirected to the login page where they can log in again. But if you'd rather they be sent to a different page, you can call `logoutSuccessFilter()` to specify a different post-logout landing page:

```
.and()  
    .logout()  
        .logoutSuccessUrl("/")
```

In this case, users will be sent to the homepage following logout.

4.3.4 Preventing cross-site request forgery

Cross-site request forgery (CSRF) is a common security attack. It involves subjecting a user to code on a maliciously designed web page that automatically (and usually secretly) submits a form to another application on behalf of a user who is often the victim of the attack. For example, a user may be presented with a form on an attacker's website that automatically posts to a URL on the user's banking website (which is presumably poorly designed and vulnerable to such an attack) to transfer money. The user may not even know that the attack happened until they notice money missing from their account.

To protect against such attacks, applications can generate a CSRF token upon displaying a form, place that token in a hidden field, and then stow it for later use on the server. When the form is submitted, the token is sent back to the server along with the rest of the form data. The request is then intercepted by the server and compared with the token that was originally generated. If the token matches, the request is allowed to proceed. Otherwise, the form must have been rendered by an evil website without knowledge of the token generated by the server.

Fortunately, Spring Security has built-in CSRF protection. Even more fortunate is that it's enabled by default and you don't need to explicitly configure it. You only

need to make sure that any forms your application submits include a field named `_csrf` that contains the CSRF token.

Spring Security even makes that easy by placing the CSRF token in a request attribute with the name `_csrf`. Therefore, you could render the CSRF token in a hidden field with the following in a Thymeleaf template:

```
<input type="hidden" name="_csrf" th:value="${_csrf.token}"/>
```

If you're using Spring MVC's JSP tag library or Thymeleaf with the Spring Security dialect, you needn't even bother explicitly including a hidden field. The hidden field will be rendered automatically for you.

In Thymeleaf, you just need to make sure that one of the attributes of the `<form>` element is prefixed as a Thymeleaf attribute. That's usually not a concern, as it's quite common to let Thymeleaf render the path as context relative. For example, the `th:action` attribute is all you need for Thymeleaf to render the hidden field for you:

```
<form method="POST" th:action="@{/login}" id="loginForm">
```

It's possible to disable CSRF support, but I'm hesitant to show you how. CSRF protection is important and easily handled in forms, so there's little reason to disable it. But if you insist on disabling it, you can do so by calling `disable()` like this:

```
.and()  
    .csrf()  
        .disable()
```

Again, I caution you not to disable CSRF protection, especially for production applications.

All of your web layer security is now configured for Taco Cloud. Among other things, you now have a custom login page and the ability to authenticate users against a JPA-backed user repository. Now let's see how you can obtain information about the logged-in user.

4.4 Knowing your user

Often, it's not enough to simply know that the user has logged in. It's usually important to also know who they are, so that you can tailor their experience.

For example, in `OrderController`, when you initially create the `Order` object that's bound to the `order` form, it'd be nice if you could prepopulate the `Order` with the user's name and address, so they don't have to reenter it for each order. Perhaps even more important, when you save their order, you should associate the `Order` entity with the `User` that created the order.

To achieve the desired connection between an `Order` entity and a `User` entity, you need to add a new property to the `Order` class:

```
@Data  
@Entity
```

```
@Table(name="Taco_Order")
public class Order implements Serializable {

    ...
    @ManyToOne
    private User user;
    ...
}
```

The `@ManyToOne` annotation on this property indicates that an order belongs to a single user, and, conversely, that a user may have many orders. (Because you're using Lombok, you won't need to explicitly define accessor methods for the property.)

In `OrderController`, the `processOrder()` method is responsible for saving an order. It will need to be modified to determine who the authenticated user is and to call `setUser()` on the `Order` object to connect the order with the user.

There are several ways to determine who the user is. These are a few of the most common ways:

- Inject a `Principal` object into the controller method
- Inject an `Authentication` object into the controller method
- Use `SecurityContextHolder` to get at the security context
- Use an `@AuthenticationPrincipal` annotated method

For example, you could modify `processOrder()` to accept a `java.security.Principal` as a parameter. You could then use the principal name to look up the user from a `UserRepository`:

```
@PostMapping
public String processOrder(@Valid Order order, Errors errors,
    SessionStatus sessionStatus,
    Principal principal) {

    ...
    User user = userRepository.findByUsername(
        principal.getName());
    order.setUser(user);
    ...
}
```

This works fine, but it litters code that's otherwise unrelated to security with security code. You can trim down some of the security-specific code by modifying `processOrder()` to accept an `Authentication` object as a parameter instead of a `Principal`:

```

@PostMapping
public String processOrder(@Valid Order order, Errors errors,
    SessionStatus sessionStatus,
    Authentication authentication) {
    ...
    User user = (User) authentication.getPrincipal();
    order.setUser(user);
    ...
}

```

With the `Authentication` in hand, you can call `getPrincipal()` to get the principal object which, in this case, is a `User`. Note that `getPrincipal()` returns a `java.util.Object`, so you need to cast it to `User`.

Perhaps the cleanest solution of all, however, is to simply accept a `User` object in `processOrder()`, but annotate it with `@AuthenticationPrincipal` so that it will be the authentication's principal:

```

@PostMapping
public String processOrder(@Valid Order order, Errors errors,
    SessionStatus sessionStatus,
    @AuthenticationPrincipal User user) {
    if (errors.hasErrors()) {
        return "orderForm";
    }
    order.setUser(user);
    orderRepo.save(order);
    sessionStatus.setComplete();
    return "redirect:/";
}

```

What's nice about `@AuthenticationPrincipal` is that it doesn't require a cast (as with `Authentication`), and it limits the security-specific code to the annotation itself. By the time you get the `User` object in `processOrder()`, it's ready to be used and assigned to the `Order`.

There's one other way of identifying who the authenticated user is, although it's a bit messy in the sense that it's very heavy with security-specific code. You can obtain an `Authentication` object from the security context and then request its principal like this:

```

Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
User user = (User) authentication.getPrincipal();

```

Although this snippet is thick with security-specific code, it has one advantage over the other approaches described: it can be used anywhere in the application, not only in a controller's handler methods. This makes it suitable for use in lower levels of the code.

Summary

- Spring Security autoconfiguration is a great way to get started with security, but most applications will need to explicitly configure security to meet their unique security requirements.
- User details can be managed in user stores backed by relational databases, LDAP, or completely custom implementations.
- Spring Security automatically protects against CSRF attacks.
- Information about the authenticated user can be obtained via the `SecurityContext` object (returned from `SecurityContextHolder.getContext()`) or injected into controllers using `@AuthenticationPrincipal`.



Working with configuration properties

This chapter covers

- Fine-tuning autoconfigured beans
- Applying configuration properties to application components
- Working with Spring profiles

Do you remember when the iPhone first came out? A small slab of metal and glass hardly fit the description of what the world had come to recognize as a phone. And yet, it pioneered the modern smartphone era, changing everything about how we communicate. Although touch phones are in many ways easier and more powerful than their predecessor, the flip phone, when the iPhone was first announced, it was hard to imagine how a device with a single button could be used to place calls.

In some ways, Spring Boot autoconfiguration is like this. Autoconfiguration greatly simplifies Spring application development. But after a decade of setting property values in Spring XML configuration and calling setter methods on bean instances, it's not immediately apparent how to set properties on beans for which there's no explicit configuration.

Fortunately, Spring Boot provides a way with configuration properties. Configuration properties are nothing more than properties on beans in the Spring

application context that can be set from one of several property sources, including JVM system properties, command-line arguments, and environment variables.

In this chapter, you’re going to take a step back from implementing new features in the Taco Cloud application to explore configuration properties. What you take away will no doubt prove useful as you move forward in the chapters that follow. We’ll start by seeing how to employ configuration properties to fine-tune what Spring Boot automatically configures.

5.1 Fine-tuning autoconfiguration

Before we dive in too deeply with configuration properties, it’s important to establish that there are two different (but related) kinds of configurations in Spring:

- *Bean wiring*—Configuration that declares application components to be created as beans in the Spring application context and how they should be injected into each other.
- *Property injection*—Configuration that sets values on beans in the Spring application context.

In Spring’s XML and Java-based configuration, these two types of configurations are often declared explicitly in the same place. In Java configuration, an @Bean-annotated method is likely to both instantiate a bean and then set values to its properties. For example, consider the following @Bean method that declares a `DataSource` for an embedded H2 database:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDataSourceBuilder()
        .setType(H2)
        .addScript("taco_schema.sql")
        .addScripts("user_data.sql", "ingredient_data.sql")
        .build();
}
```

Here the `addScript()` and `addScripts()` methods set some `String` properties with the name of SQL scripts that should be applied to the database once the data source is ready. Whereas this is how you might configure a `DataSource` bean if you aren’t using Spring Boot, autoconfiguration makes this method completely unnecessary.

If the H2 dependency is available in the run-time classpath, then Spring Boot automatically creates an appropriate `DataSource` bean in the Spring application context. The bean applies the SQL scripts `schema.sql` and `data.sql`.

But what if you want to name the SQL scripts something else? Or what if you need to specify more than two SQL scripts? That’s where configuration properties come in. But before you can start using configuration properties, you need to understand where those properties come from.

5.1.1 Understanding Spring's environment abstraction

The Spring environment abstraction is a one-stop shop for any configurable property. It abstracts the origins of properties so that beans needing those properties can consume them from Spring itself. The Spring environment pulls from several property sources, including

- JVM system properties
- Operating system environment variables
- Command-line arguments
- Application property configuration files

It then aggregates those properties into a single source from which Spring beans can be injected. Figure 5.1 illustrates how properties from property sources flow through the Spring environment abstraction to Spring beans.

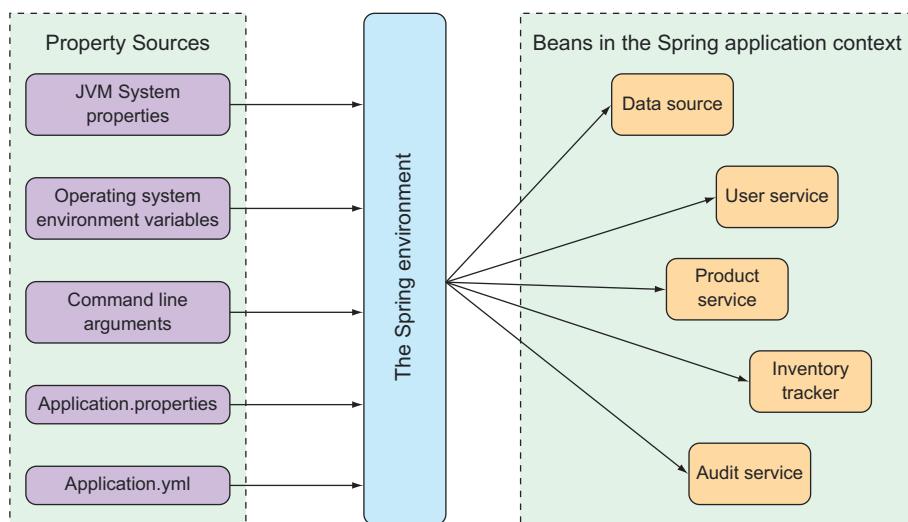


Figure 5.1 The Spring environment pulls properties from property sources and makes them available to beans in the application context.

The beans that are automatically configured by Spring Boot are all configurable by properties drawn from the Spring environment. As a simple example, suppose that you would like the application's underlying servlet container to listen for requests on some port other than the default port of 8080. To do that, specify a different port by setting the `server.port` property in `src/main/resources/application.properties` like this:

```
server.port=9090
```

Personally, I prefer using YAML when setting configuration properties. Therefore, instead of using application.properties, I might set the server.port value in src/main/resources/application.yml like this:

```
server:  
  port: 9090
```

If you'd prefer to configure that property externally, you could also specify the port when starting the application using a command-line argument:

```
$ java -jar tacocloud-0.0.5-SNAPSHOT.jar --server.port=9090
```

If you want the application to always start on a specific port, you could set it one time as an operating system environment variable:

```
$ export SERVER_PORT=9090
```

Notice that when setting properties as environment variables, the naming style is slightly different to accommodate restrictions placed on environment variable names by the operating system. That's OK. Spring is able to sort it out and interpret SERVER_PORT as server.port with no problems.

As I said, there are several ways of setting configuration properties. And when we get to chapter 14, you'll see yet another way of setting configuration properties in a centralized configuration server. In fact, there are several hundred configuration properties you can use to tweak and adjust how Spring beans behave. You've already seen a few: server.port in this chapter and security.user.name and security.user.password in the previous chapter.

It's impossible to examine all of the available configuration properties in this chapter. Even so, let's take a look at a few of the most useful configuration properties you might commonly encounter. We'll start with a few properties that let you tweak the autoconfigured data source.

5.1.2 Configuring a data source

At this point, the Taco Cloud application is still unfinished, but you'll have several more chapters to take care of that before you're ready to deploy the application. As such, the embedded H2 database you're using as a data source is perfect for your needs—for now. But once you take the application into production, you'll probably want to consider a more permanent database solution.

Although you could explicitly configure your own DataSource bean, that's usually unnecessary. Instead, it's simpler to configure the URL and credentials for your database via configuration properties. For example, if you were to start using a MySQL database, you might add the following configuration properties to application.yml:

```
spring:  
  datasource:  
    url: jdbc:mysql://localhost/tacocloud
```

```
username: tacodb
password: tacopassword
```

Although you'll need to add the appropriate JDBC driver to the build, you won't usually need to specify the JDBC driver class; Spring Boot can figure it out from the structure of the database URL. But if there's a problem, you can try setting the `spring.datasource.driver-class-name` property:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacodb
    password: tacopassword
    driver-class-name: com.mysql.jdbc.Driver
```

Spring Boot uses this connection data when autoconfiguring the `DataSource` bean. The `DataSource` bean will be pooled using Tomcat's JDBC connection pool if it's available on the classpath. If not, Spring Boot looks for and uses one of these other connection pool implementations on the classpath:

- HikariCP
- Commons DBCP 2

Although these are the only connection pool options available through autoconfiguration, you're always welcome to explicitly configure a `DataSource` bean to use whatever connection pool implementation you'd like.

Earlier in this chapter, we suggested that there might be a way to specify the database initialization scripts to run when the application starts. In that case, the `spring.datasource.schema` and `spring.datasource.data` properties prove useful:

```
spring:
  datasource:
    schema:
      - order-schema.sql
      - ingredient-schema.sql
      - taco-schema.sql
      - user-schema.sql
    data:
      - ingredients.sql
```

Maybe explicit data source configuration isn't your style. Instead, perhaps you'd prefer to configure your data source in JNDI and have Spring look it up from there. In that case, set up your data source by configuring `spring.datasource.jndi-name`:

```
spring:
  datasource:
    jndi-name: java:/comp/env/jdbc/tacoCloudDS
```

If you set the `spring.datasource.jndi-name` property, the other data source connection properties (if set) are ignored.

5.1.3 Configuring the embedded server

You've already seen how to set the servlet container's port by setting `server.port`. What I didn't show you is what happens if `server.port` is set to 0:

```
server:  
  port: 0
```

Although you're explicitly setting `server.port` to 0, the server won't start on port 0. Instead, it'll start on a randomly chosen available port. This is useful when running automated integration tests to ensure that any concurrently running tests don't clash on a hard-coded port number. As you'll see in chapter 13, it's also useful when you don't care what port your application starts on because it's a microservice that will be looked up from a service registry.

But there's more to the underlying server than just a port. One of the most common things you'll need to do with the underlying container is to set it up to handle HTTPS requests. To do that, the first thing you must do is create a keystore using the JDK's `keytool` command-line utility:

```
$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

You'll be asked several questions about your name and organization, most of which are irrelevant. But when asked for a password, remember what you choose. For the sake of this example, I chose `letmein` as the password.

Next, you'll need to set a few properties to enable HTTPS in the embedded server. You could specify them all on the command line, but that would be terribly inconvenient. Instead, you'll probably set them in the file's `application.properties` or `application.yml`. In `application.yml`, the properties might look like this:

```
server:  
  port: 8443  
  ssl:  
    key-store: file:///path/to/mykeys.jks  
    key-store-password: letmein  
    key-password: letmein
```

Here the `server.port` property is set to 8443, a common choice for development HTTPS servers. The `server.ssl.key-store` property should be set to the path where the keystore file is created. Here it's shown with a `file://` URL to load it from the filesystem, but if you package it within the application JAR file, you'll use a `classpath:` URL to reference it. And both the `server.ssl.key-store-password` and `server.ssl.key-password` properties are set to the password that was given when creating the keystore.

With these properties in place, your application should be listening for HTTPS requests on port 8443. Depending on which browser you're using, you may encounter a warning about the server not being able to verify its identity. This is nothing to worry about when serving from localhost during development.

5.1.4 Configuring logging

Most applications provide some form of logging. And even if your application doesn't log anything directly, the libraries that your application uses will certainly log their activity.

By default, Spring Boot configures logging via Logback (<http://logback.qos.ch>) to write to the console at an INFO level. You've probably already seen plenty of INFO-level entries in the application logs as you've run the application and other examples.

For full control over the logging configuration, you can create a logback.xml file at the root of the classpath (in src/main/resources). Here's an example of a simple logback.xml file you might use:

```
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>
                %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
            </pattern>
        </encoder>
    </appender>
    <logger name="root" level="INFO"/>
    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

Aside from the pattern used for logging, this Logback configuration is more or less equivalent to the default you'll get if you have no logback.xml file. But by editing logback.xml you can gain full control over your application's log files.

NOTE The specifics of what can go into logback.xml are outside the scope of this book. Refer to Logback's documentation for more information.

The most common changes you'll make to a logging configuration are to change the logging levels and perhaps to specify a file where the logs should be written. With Spring Boot configuration properties, you can make those changes without having to create a logback.xml file.

To set the logging levels, you create properties that are prefixed with `logging.level`, followed by the name of the logger for which you want to set the logging level. For instance, suppose you'd like to set the root logging level to WARN, but log Spring Security logs at a DEBUG level. The following entries in application.yml will take care of that for you:

```
logging:
  level:
    root: WARN
    org:
      springframework: DEBUG
      security: DEBUG
```

Optionally, you can collapse the Spring Security package name to a single line for easier reading:

```
logging:  
  level:  
    root: WARN  
    org.springframework.security: DEBUG
```

Now suppose that you want to write the log entries to the file TacoCloud.log at /var/logs/. The logging.path and logging.file properties can help achieve that:

```
logging:  
  path: /var/logs/  
  file: TacoCloud.log  
  level:  
    root: WARN  
    org:  
      springframework:  
        security: DEBUG
```

Assuming that the application has write permissions to /var/logs/, the log entries will be written to /var/logs/TacoCloud.log. By default, the log files rotate once they reach 10 MB in size.

5.1.5 Using special property values

When setting properties, you aren't limited to declaring their values as hard-coded String and numeric values. Instead, you can derive their values from other configuration properties.

For example, suppose (for whatever reason) you want to set a property named greeting.welcome to echo the value of another property named spring.application.name. To achieve this, you could use the \${} placeholder markers when setting greeting.welcome:

```
greeting:  
  welcome: ${spring.application.name}
```

You can even embed that placeholder amidst other text:

```
greeting:  
  welcome: You are using ${spring.application.name}.
```

As you've seen, configuring Spring's own components with configuration properties makes it easy to inject values into those components' properties and to fine-tune autoconfiguration. Configuration properties aren't exclusive to the beans that Spring creates. With a small amount of effort, you can take advantage of configuration properties in your own beans. Let's see how.

5.2 Creating your own configuration properties

As I mentioned earlier, configuration properties are nothing more than properties of beans that have been designated to accept configurations from Spring's environment abstraction. What I didn't mention is how those beans are designated to consume those configurations.

To support property injection of configuration properties, Spring Boot provides the `@ConfigurationProperties` annotation. When placed on any Spring bean, it specifies that the properties of that bean can be injected from properties in the Spring environment.

To demonstrate how `@ConfigurationProperties` works, suppose that you've added the following method to `OrderController` to list the authenticated user's past orders:

```
@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user));

    return "orderList";
}
```

Along with that, you've also added the necessary `findByUser()` method to `OrderRepository`:

```
List<Order> findByUserOrderByPlacedAtDesc(User user);
```

Notice that this repository method is named with a clause of `OrderByPlacedAtDesc`. The `OrderBy` portion specifies a property by which the results will be ordered—in this case, the `placedAt` property. The `Desc` at the end causes the ordering to be in descending order. Therefore, the list of orders returned will be sorted most recent to least recent.

As written, this controller method may be useful after the user has placed a handful of orders. But it could become a bit unwieldy for the most avid of taco connoisseurs. A few orders displayed in the browser are useful; a never-ending list of hundreds of orders is just noise. Let's say that you want to limit the number of orders displayed to the most recent 20 orders. You can change `ordersForUser()`

```
@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

    Pageable pageable = PageRequest.of(0, 20);
    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

    return "orderList";
}
```

along with the corresponding changes to OrderRepository:

```
List<Order> findByUserOrderByPlacedAtDesc(  
    User user, Pageable pageable);
```

Here you've changed the signature of the `findByUserOrderByPlacedAtDesc()` method to accept a `Pageable` as a parameter. `Pageable` is Spring Data's way of selecting some subset of the results by a page number and page size. In the `ordersForUser()` controller method, you constructed a `PageRequest` object that implemented `Pageable` to request the first page (page zero) with a page size of 20 to get up to 20 of the most recently placed orders for the user.

Although this works fantastically, it leaves me a bit uneasy that you've hard-coded the page size. What if you later decide that 20 is too many orders to list, and you decide to change it to 10? Because it's hard-coded, you'd have to rebuild and redeploy the application.

Rather than hardcode the page size, you can set it with a custom configuration property. First, you need to add a new property called `pageSize` to `OrderController` and then annotate `OrderController` with `@ConfigurationProperties` as shown in the next listing.

Listing 5.1 Enabling configuration properties in OrderController

```
@Controller  
@RequestMapping("/orders")  
@SessionAttributes("order")  
@ConfigurationProperties(prefix="taco.orders")  
public class OrderController {  
  
    private int pageSize = 20;  
  
    public void setPageSize(int pageSize) {  
        this.pageSize = pageSize;  
    }  
  
    ...  
  
    @GetMapping  
    public String ordersForUser(  
        @AuthenticationPrincipal User user, Model model) {  
  
        Pageable pageable = PageRequest.of(0, pageSize);  
        model.addAttribute("orders",  
            orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));  
  
        return "orderList";  
    }  
}
```

The most significant change made in listing 5.1 is the addition of the `@ConfigurationProperties` annotation. Its `prefix` attribute is set to `taco.orders`, which means that

when setting the `pageSize` property, you need to use a configuration property named `taco.orders.pageSize`.

The new `pageSize` property defaults to 20. But you can easily change it to any value you want by setting a `taco.orders.pageSize` property. For example, you could set this property in `application.yml` like this:

```
taco:
  orders:
    pageSize: 10
```

Or, if you need to make a quick change while in production, you can do so without having to rebuild and redeploy the application by setting the `taco.orders.pageSize` property as an environment variable:

```
$ export TACO_ORDERS_PAGESIZE=10
```

Any means by which a configuration property can be set can be used to adjust the page size of the recent orders page. Next, we'll look at how to set configuration data in property holders.

5.2.1 Defining configuration properties holders

There's nothing that says `@ConfigurationProperties` must be set on a controller or any other specific kind of bean. `@ConfigurationProperties` are in fact often placed on beans whose sole purpose in the application is to be holders of configuration data. This keeps configuration-specific details out of the controllers and other application classes. It also makes it easy to share common configuration properties among several beans that may make use of that information.

In the case of the `pageSize` property in `OrderController`, you could extract it to a separate class. The following listing uses the `OrderProps` class in such a way.

Listing 5.2 Extracting pageSize to a holder class

```
package tacos.web;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;
import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
public class OrderProps {

    private int pageSize = 20;

}
```

As you did with `OrderController`, the `pageSize` property defaults to 20 and `OrderProps` is annotated with `@ConfigurationProperties` to have a prefix of `taco.orders`.

It's also annotated with `@Component` so that Spring component scanning will automatically discover it and create it as a bean in the Spring application context. This is important, as the next step is to inject the `OrderProps` bean into `OrderController`.

There's nothing particularly special about configuration property holders. They're beans that have their properties injected from the Spring environment. They can be injected into any other bean that needs those properties. For `OrderController`, this means removing the `pageSize` property from `OrderController` and instead injecting and using the `OrderProps` bean:

```
@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
public class OrderController {

    private OrderRepository orderRepo;

    private OrderProps props;

    public OrderController(OrderRepository orderRepo,
                          OrderProps props) {
        this.orderRepo = orderRepo;
        this.props = props;
    }

    ...

    @GetMapping
    public String ordersForUser(
        @AuthenticationPrincipal User user, Model model) {

        Pageable pageable = PageRequest.of(0, props.getPageSize());
        model.addAttribute("orders",
                           orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

        return "orderList";
    }

    ...
}
```

Now `OrderController` is no longer responsible for handling its own configuration properties. This keeps the code in `OrderController` slightly neater and allows you to reuse the properties in `OrderProps` in any other bean that may need them. Moreover, you're collecting configuration properties that pertain to orders in one place: the `OrderProps` class. If you need to add, remove, rename, or otherwise change the properties therein, you only need to apply those changes in `OrderProps`.

For example, let's pretend that you're using the `pageSize` property in several other beans when you decide it would be best to apply some validation to that property to limit its values to no less than 5 and no more than 25. Without a holder bean, you'd

have to apply validation annotations to OrderController, the pageSize property, and all other classes using that property. But because you've extracted pageSize into OrderProps, you only must make the changes to OrderProps:

```
package tacos.web;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;
import org.springframework.validation.annotation.Validated;
import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
@Validated
public class OrderProps {

    @Min(value=5, message="must be between 5 and 25")
    @Max(value=25, message="must be between 5 and 25")
    private int pageSize = 20;

}
//end::validated[]
```

Although you could as easily apply the @Validated, @Min, and @Max annotations to OrderController (and any other beans that can be injected with OrderProps), it would just clutter up OrderController that much more. With a configuration property holder bean, you've collected configuration property specifics in one place, leaving the classes that need those properties relatively clean.

5.2.2 Declaring configuration property metadata

Depending on your IDE, you may have noticed that the taco.orders.pageSize entry in application.yml (or application.properties) has a warning saying something like Unknown Property ‘taco’. This warning appears because there's missing metadata concerning the configuration property you just created. Figure 5.2 shows what this looks like when I hover over the taco portion of the property in the Spring Tool Suite.

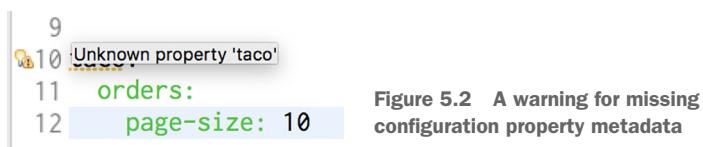


Figure 5.2 A warning for missing configuration property metadata

Configuration property metadata is completely optional and doesn't prevent configuration properties from working. But the metadata can be useful for providing some minimal documentation around the configuration properties, especially in the IDE.

For example, when I hover over the `security.user.password` property, I see what's shown in figure 5.3. Although the hover help you get is minimal, it can be enough to help understand what the property is used for and how to use it.

```

1 security:
2   user:
3     name: buzz
4     password: infinity
5
6 taco:
7   or Password for the default user name.
8
9
10 management:

```

Figure 5.3 Hover documentation for configuration properties in the Spring Tool Suite

To help those who might use the configuration properties that you define—which might even be you—it's generally a good idea to create some metadata around those properties. At least it gets rid of those annoying yellow warnings in the IDE.

To create metadata for your custom configuration properties, you'll need to create a file under the META-INF (for example, in the project under `src/main/resources/META-INF`) named `additional-spring-configuration-metadata.json`.

QUICK-FIXING MISSING METADATA.

If you're using the Spring Tool Suite, there's a quick-fix option for creating missing property metadata. Place your cursor on the line with the missing metadata warning and open the quick-fix pop up with CMD-1 on Mac or Ctrl-1 on Windows and Linux (see figure 5.4).

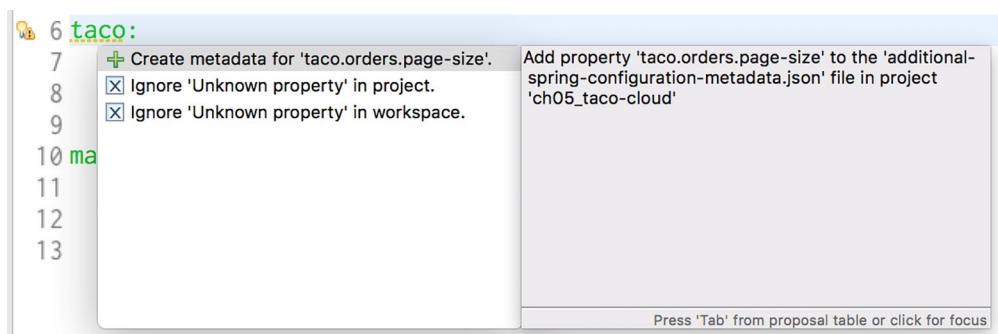


Figure 5.4 Creating configuration property metadata with the quick-fix pop up in Spring Tool Suite

Then select the Create Metadata for ... option to add some metadata for the property (in `additional-spring-configuration-metadata.json` as this figure shows) and create that file if it doesn't already exist.

For the `taco.orders.pageSize` property, you can set up the metadata with the following JSON:

```
{
  "properties": [
    {
      "name": "taco.orders.page-size",
      "type": "java.lang.String",
      "description":
        "Sets the maximum number of orders to display in a list."
    }
  ]
}
```

Notice that the property name referenced in the metadata is `taco.orders.pageSize`. Spring Boot's flexible property naming allows for variations in property names such that `taco.orders.page-size` is equivalent to `taco.orders.pageSize`.

With that metadata in place, the warnings should be gone. What's more, if you hover over the `taco.orders.pageSize` property, you'll see the description shown in figure 5.5.

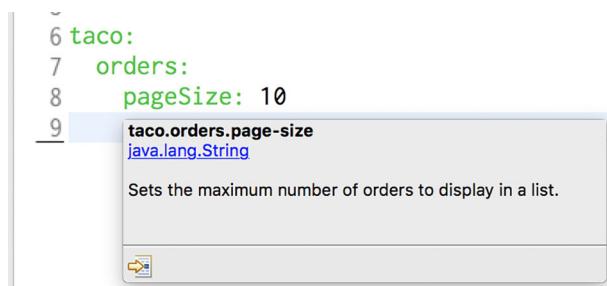


Figure 5.5 Hover help for custom configuration properties

Also, as shown in figure 5.6, you get autocompletion help from the IDE, just like Spring-provided configuration properties.

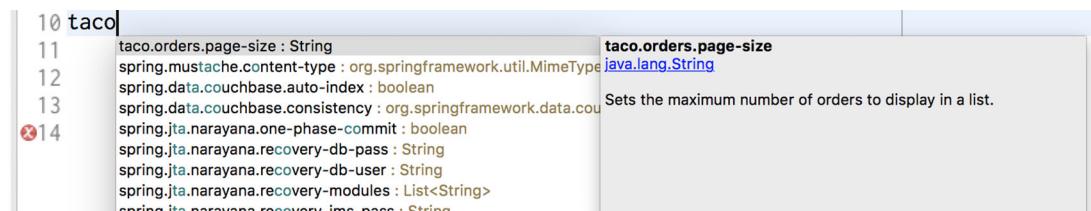


Figure 5.6 Configuration property metadata enables autocompletion of properties.

As you've seen, configuration properties are useful for tweaking both autoconfigured components as well as the details injected into your own application beans. But

what if you need to configure different properties for different deployment environments? Let's take a look at how to use Spring profiles to set up environment-specific configuration.

5.3 Configuring with profiles

When applications are deployed to different run-time environments, usually some configuration details differ. The details of a database connection, for instance, are likely not the same in a development environment as in a quality assurance environment, and different still in a production environment. One way to configure properties uniquely in one environment over another is to use environment variables to specify configuration properties instead of defining them in application.properties and application.yml.

For instance, during development you can lean on the autoconfigured embedded H2 database. But in production you can set database configuration properties as environment variables like this:

```
% export SPRING_DATASOURCE_URL=jdbc:mysql://localhost/tacocloud  
% export SPRING_DATASOURCE_USERNAME=tacouser  
% export SPRING_DATASOURCE_PASSWORD=tacopassword
```

Although this will work, it's somewhat cumbersome to specify more than one or two configuration properties as environment variables. Moreover, there's no good way to track changes to environment variables or to easily roll back changes if there's a mistake.

Instead, I prefer to take advantage of Spring profiles. Profiles are a type of conditional configuration where different beans, configuration classes, and configuration properties are applied or ignored based on what profiles are active at runtime.

For instance, let's say that for development and debugging purposes, you want to use the embedded H2 database, and you want the logging levels for the Taco Cloud code to be set to DEBUG. But in production, you want to use an external MySQL database and set the logging levels to WARN. In the development situation, it's easy enough to not set any data-source properties and get the autoconfigured H2 database. And as for debug-level logging, you can set the logging.level.tacos property for the tacos base package to DEBUG in application.yml:

```
logging:  
  level:  
    tacos: DEBUG
```

This is precisely what you need for development purposes. But if you were to deploy this application in a production setting with no further changes to application.yml, you'd still have debug logging for the tacos package and an embedded H2 database. What you need is to define a profile with properties suited for production.

5.3.1 Defining profile-specific properties

One way to define profile-specific properties is to create yet another YAML or properties file containing only the properties for production. The name of the file should follow this convention: application-{profile name}.yml or application-{profile name}.properties. Then you can specify the configuration properties appropriate to that profile. For example, you could create a new file named application-prod.yml that contains the following properties:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword
  logging:
    level:
      tacos: WARN
```

Another way to specify profile-specific properties works only with YAML configuration. It involves placing profile-specific properties alongside non-profiled properties in application.yml, separated by three hyphens and the `spring.profiles` property to name the profile. When applying the production properties to application.yml in this way, the entire application.yml would look like this:

```
logging:
  level:
    tacos: DEBUG

---
spring:
  profiles: prod

  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword

  logging:
    level:
      tacos: WARN
```

As you can see, this application.yml file is divided into two sections by a set of triple hyphens (---). The second section specifies a value for `spring.profiles`, indicating that the properties that follow apply to the `prod` profile. The first section, on the other hand, doesn't specify a value for `spring.profiles`. Therefore, its properties are common to all profiles or are defaults if the active profile doesn't otherwise have the properties set.

Regardless of which profiles are active when the application runs, the logging level for the `tacos` package will be set to `DEBUG` by the property set in the default profile. But if the profile named `prod` is active, then the `logging.level.tacos` property will

be overridden with `WARN`. Likewise, if the `prod` profile is active, then the data-source properties will be set to use the external MySQL database.

You can define properties for as many profiles as you need by creating additional YAML or properties files named with the pattern `application-{profile name}.yml` or `application-{profile name}.properties`. Or, if you prefer, type three more dashes in `application.yml` along with another `spring.profiles` property to specify the profile name. Then add all of the profile-specific properties you need.

5.3.2 Activating profiles

Setting profile-specific properties will do no good unless those profiles are active. But how can you make a profile active? All it takes to make a profile active is to include it in the list of profile names given to the `spring.profiles.active` property. For example, you could set it in `application.yml` like this:

```
spring:  
  profiles:  
    active:  
      - prod
```

But that's perhaps the worst possible way to set an active profile. If you set the active profile in `application.yml`, then that profile becomes the default profile, and you achieve none of the benefits of using profiles to separate the production-specific properties from development properties. Instead, I recommend that you set the active profile(s) with environment variables. On the production environment, you would set `SPRING_PROFILES_ACTIVE` like this:

```
% export SPRING_PROFILES_ACTIVE=prod
```

From then on, any applications deployed to that machine will have the `prod` profile active and the corresponding configuration properties would take precedence over the properties in the default profile.

If you're running the application as an executable JAR file, you might also set the active profile with a command-line argument like this:

```
% java -jar taco-cloud.jar --spring.profiles.active=prod
```

Note that the `spring.profiles.active` property name contains the plural word `profiles`. This means you can specify more than one active profile. Often, this is with a comma-separated list as when setting it with an environment variable:

```
% export SPRING_PROFILES_ACTIVE=prod,audit,ha
```

But in YAML, you'd specify it as a list like this:

```
spring:  
  profiles:  
    active:
```

- prod
- audit
- ha

It's also worth noting that if you deploy a Spring application to Cloud Foundry, a profile named `cloud` is automatically activated for you. If Cloud Foundry is your production environment, you'll want to be sure to specify production-specific properties under the `cloud` profile.

As it turns out, profiles aren't useful only for conditionally setting configuration properties in a Spring application. Let's see how to declare beans specific to an active profile.

5.3.3 Conditionally creating beans with profiles

Sometimes it's useful to provide a unique set of beans for different profiles. Normally, any bean declared in a Java configuration class is created, regardless of which profile is active. But suppose there are some beans that you only need to be created if a certain profile is active. In that case, the `@Profile` annotation can designate beans as only being applicable to a given profile.

For instance, you have a `CommandLineRunner` bean declared in `TacoCloudApplication` that's used to load the embedded database with ingredient data when the application starts. That's great for development, but would be unnecessary (and undesirable) in a production application. To prevent the ingredient data from being loaded every time the application starts in a production deployment, you could annotate the `CommandLineRunner` bean method with `@Profile` like this:

```
@Bean
@Profile("dev")
public CommandLineRunner dataLoader(IngredientRepository repo,
    UserRepository userRepo, PasswordEncoder encoder) {
    ...
}
```

Or suppose that you need the `CommandLineRunner` created if either the `dev` profile or `qa` profile is active. In that case, you can list the profiles for which the bean should be created:

```
@Bean
@Profile({"dev", "qa"})
public CommandLineRunner dataLoader(IngredientRepository repo,
    UserRepository userRepo, PasswordEncoder encoder) {
    ...
}
```

Now the ingredient data will only be loaded if the `dev` or `qa` profiles are active. That would mean that you'd need to activate the `dev` profile when running the application

in the development environment. It would be even more convenient if that CommandLineRunner bean were always created unless the prod profile is active. In that case, you can apply @Profile like this:

```
@Bean  
@Profile("!prod")  
public CommandLineRunner dataLoader(IngredientRepository repo,  
        UserRepository userRepo, PasswordEncoder encoder) {  
  
    ...  
}
```

Here, the exclamation mark (!) negates the profile name. Effectively, it states that the CommandLineRunner bean will be created if the prod profile isn't active.

It's also possible to use @Profile on an entire @Configuration-annotated class. For example, suppose that you were to extract the CommandLineRunner bean into a separate configuration class named `DevelopmentConfig`. Then you could annotate `DevelopmentConfig` with @Profile:

```
@Profile({"!prod", "!qa"})  
@Configuration  
public class DevelopmentConfig {  
  
    @Bean  
    public CommandLineRunner dataLoader(IngredientRepository repo,  
            UserRepository userRepo, PasswordEncoder encoder) {  
  
        ...  
    }  
}
```

Here, the CommandLineRunner bean (as well as any other beans defined in `DevelopmentConfig`) will only be created if neither the prod nor qa profiles are active.

Summary

- Spring beans can be annotated with `@ConfigurationProperties` to enable injection of values from one of several property sources.
- Configuration properties can be set in command-line arguments, environment variables, JVM system properties, properties files, or YAML files, among other options.
- Configuration properties can be used to override autoconfiguration settings, including the ability to specify a data-source URL and logging levels.
- Spring profiles can be used with property sources to conditionally set configuration properties based on the active profile(s).

Part 2

Integrated Spring

T

he chapters in part 2 cover topics that help integrate your Spring application with other applications.

Chapter 6 expands on the discussion of Spring MVC started in chapter 2 by looking at how to write REST APIs in Spring. We'll look at how to define REST endpoints in Spring MVC, enable hyperlinked REST resources, and automatically generate repository-based REST endpoints with Spring Data REST. Chapter 7 switches perspective to show how a Spring application can consume a REST API. In chapter 8, we'll look at using asynchronous communication to enable a Spring application to both send and receive messages using the Java Message Service (JMS), RabbitMQ, and Kafka. And finally, chapter 9 discusses declarative application integration using the Spring Integration project. We'll cover processing data in real time, defining integration flows, and integrating with external systems like emails and filesystems.

Creating REST services

This chapter covers

- Defining REST endpoints in Spring MVC
- Enabling hyperlinked REST resources
- Automatic repository-based REST endpoints

“The web browser is dead. What now?”

Roughly a dozen years ago, I heard someone suggest that the web browser was nearing legacy status and that something else would take over. But how could this be? What could possibly dethrone the near-ubiquitous web browser? How would we consume the growing number of sites and online services if not with a web browser? Surely these were the ramblings of a madman!

Fast-forward to the present day and it’s clear that the web browser hasn’t gone away. But it no longer reigns as the primary means of accessing the internet. Mobile devices, tablets, smart watches, and voice-based devices are now commonplace. And even many browser-based applications are actually running JavaScript applications rather than letting the browser be a dumb terminal for server-rendered content.

With such a vast selection of client-side options, many applications have adopted a common design where the user interface is pushed closer to the client and the

server exposes an API through which all kinds of clients can interact with the backend functionality.

In this chapter, you’re going to use Spring to provide a REST API for the Taco Cloud application. You’ll use what you learned about Spring MVC in chapter 2 to create RESTful endpoints with Spring MVC controllers. You’ll also automatically expose REST endpoints for the Spring Data repositories you defined in chapter 4. Finally, we’ll look at ways to test and secure those endpoints.

But first, you’ll start by writing a few new Spring MVC controllers that expose backend functionality with REST endpoints to be consumed by a rich web frontend.

6.1 Writing RESTful controllers

I hope you don’t mind, but while you were turning the page and reading the introduction to this chapter, I took it upon myself to reimagine the user interface for Taco Cloud. What you’ve been working with has been fine for getting started, but it lacked in the aesthetics department.

Figure 6.1 is just a sample of what the new Taco Cloud looks like. Pretty snazzy, huh?

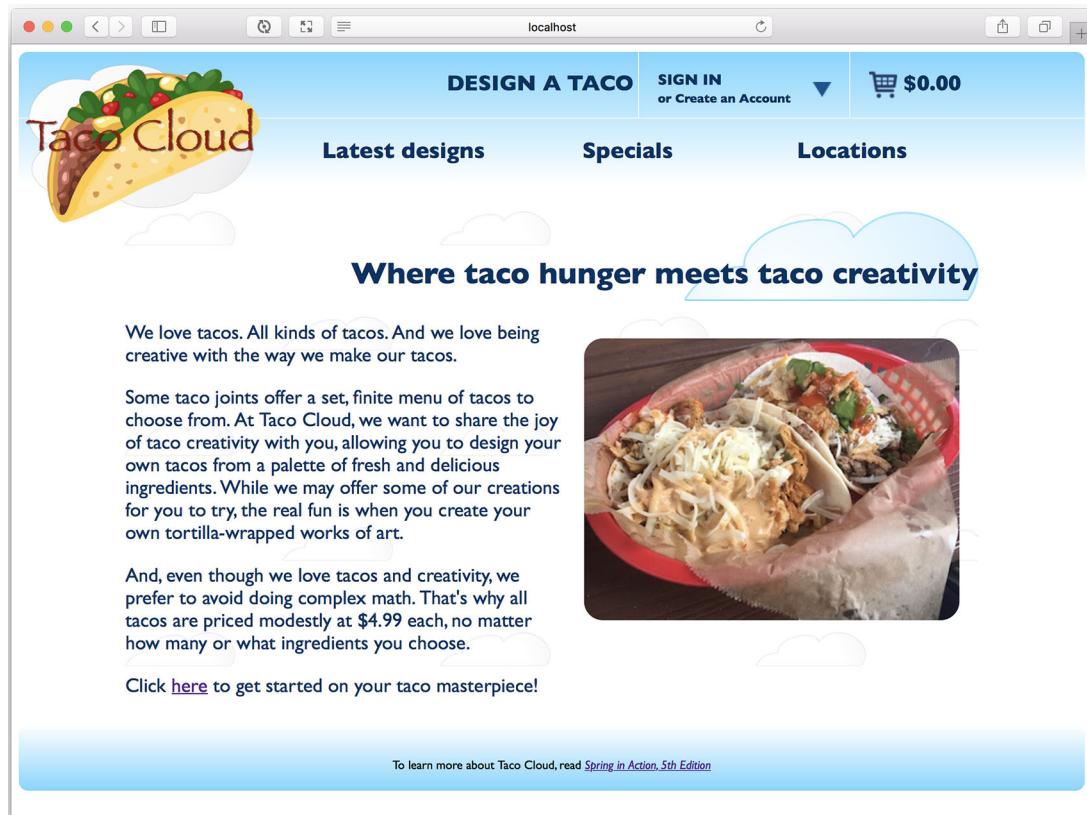


Figure 6.1 The new Taco Cloud home page

And while I was spiffing up the Taco Cloud look, I also decided to build the frontend as a single-page application using the popular Angular framework. Ultimately, this new browser UI will replace the server-rendered pages you created in chapter 2. But for that to work, you'll need to create a REST API that the Angular-based¹ UI will communicate with to save and fetch taco data.

To SPA or not to SPA?

You developed a traditional multipage application (MPA) with Spring MVC in chapter 2, and now you're replacing that with a single-page application (SPA) based on Angular. But I'm not suggesting that SPA is always a better choice than MPA.

Because presentation is largely decoupled from backend processing in a SPA, it affords the opportunity to develop more than one user interface (such as a native mobile application) for the same backend functionality. It also opens up the opportunity for integration with other applications that can consume the API. But not all applications require that flexibility, and MPA is a simpler design if all you need is to display information on a web page.

This isn't a book on Angular, so the code in this chapter will focus primarily on the backend Spring code. I'll show just enough Angular code to give you a feel for how the client side works. Rest assured that the complete set of code, including the Angular frontend, is available as part of the downloadable code for the book and at <https://github.com/habuma/spring-in-action-5-samples>. You may also be interested in reading *Angular in Action* by Jeremy Wilken (Manning, 2018) and *Angular Development with TypeScript, Second Edition* by Yakov Fain and Anton Moiseev (Manning, 2018).

In a nutshell, the Angular client code will communicate with an API that you'll create throughout this chapter by way of HTTP requests. In chapter 2 you used `@GetMapping` and `@PostMapping` annotations to fetch and post data to the server. Those same annotations will still come in handy as you define your REST API. In addition, Spring MVC supports a handful of other annotations for various types of HTTP requests, as listed in table 6.1.

Table 6.1 Spring MVC's HTTP request-handling annotations

Annotation	HTTP method	Typical use ^a
<code>@GetMapping</code>	HTTP GET requests	Reading resource data
<code>@PostMapping</code>	HTTP POST requests	Creating a resource
<code>@PutMapping</code>	HTTP PUT requests	Updating a resource
<code>@PatchMapping</code>	HTTP PATCH requests	Updating a resource

¹ I chose to use Angular, but the choice of frontend framework should have little to no bearing on how the backend Spring code is written. Feel free to choose Angular, React, Vue.js, or whatever frontend technology suits you best.

Table 6.1 Spring MVC's HTTP request-handling annotations (*continued*)

Annotation	HTTP method	Typical use ^a
@DeleteMapping	HTTP DELETE requests	Deleting a resource
@RequestMapping	General purpose request handling; HTTP method specified in the method attribute	

^a Mapping HTTP methods to create, read, update, and delete (CRUD) operations isn't a perfect match, but in practice, that's how they're often used and how you'll use them in Taco Cloud.

To see these annotations in action, you'll start by creating a simple REST endpoint that fetches a few of the most recently created tacos.

6.1.1 *Retrieving data from the server*

One of the coolest things about Taco Cloud is that it allows taco fanatics to design their own taco creations and share them with their fellow taco lovers. To this end, Taco Cloud needs to be able to display a list of the most recently created tacos when the Latest Designs link is clicked.

In the Angular code I've defined a `RecentTacosComponent` that will display the most recently created tacos. The complete TypeScript code for `RecentTacosComponent` is shown in the next listing.

Listing 6.1 Angular component for displaying recent tacos

```
import { Component, OnInit, Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'recent-tacos',
  templateUrl: 'recents.component.html',
  styleUrls: ['./recents.component.css']
})

@Injectable()
export class RecentTacosComponent implements OnInit {
  recentTacos: any;

  constructor(private httpClient: HttpClient) { }

  ngOnInit() {
    this.httpClient.get('http://localhost:8080/design/recent')
      .subscribe(data => this.recentTacos = data);
  }
}
```

Fetches
recent tacos
from the
server

Turn your attention to the `ngOnInit()` method. In that method, `RecentTacosComponent` uses the injected `Http` module to perform an HTTP GET request to `http://localhost:8080/design/recent`, expecting that the response will contain a list of taco

designs, which will be placed in the recentTacos model variable. The view (in recentTacos.html) will present that model data as HTML to be rendered in the browser. The end result might look something like figure 6.2, after three tacos have been created.

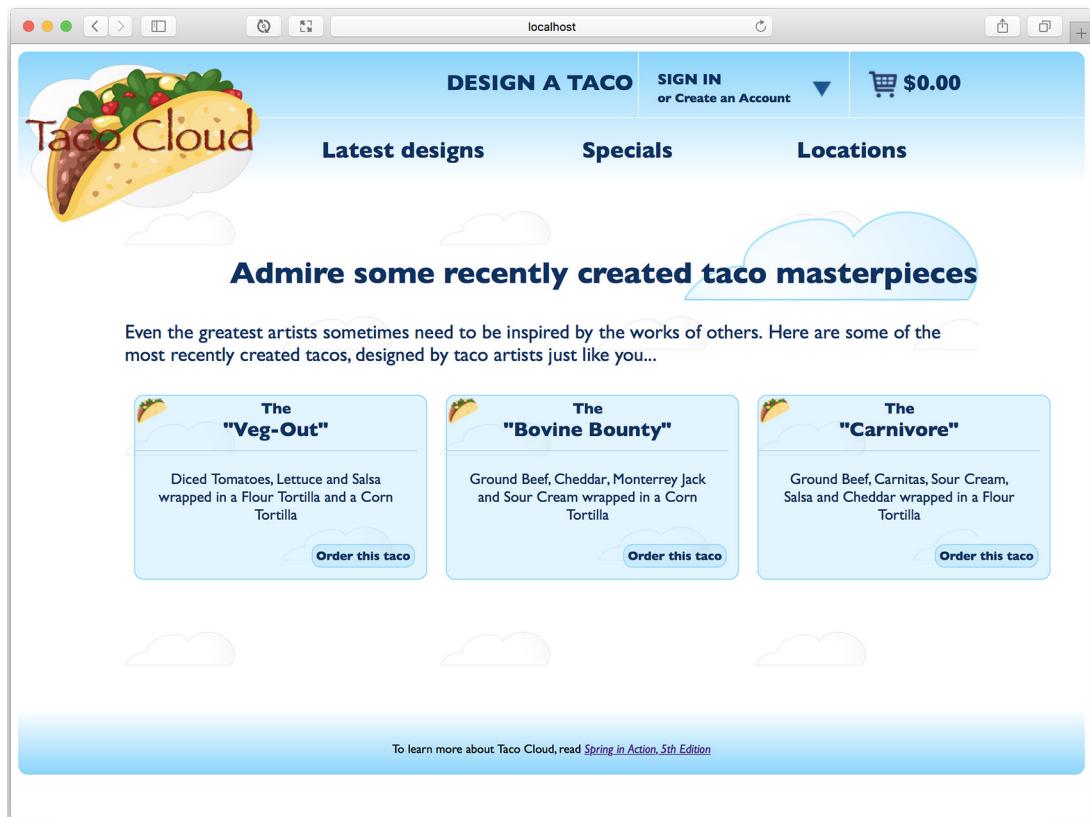


Figure 6.2 Displaying the most recently created tacos

The missing piece in this puzzle is an endpoint that handles GET requests for /design/recent and responds with a list of recently designed tacos. You'll create a new controller to handle such a request. The next listing shows the controller for the job.

Listing 6.2 A RESTful controller for taco design API requests

```
package tacos.web.api;  
  
import java.util.Optional;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.domain.PageRequest;  
import org.springframework.data.domain.Sort;
```

```

import org.springframework.hateoas.EntityLinks;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import tacos.Taco;
import tacos.data.TacoRepository;

@RestController
@RequestMapping(path="/design",
    produces="application/json")
@CrossOrigin(origins="*")
public class DesignTacoController {
    private TacoRepository tacoRepo;

    @Autowired
    EntityLinks entityLinks;

    public DesignTacoController(TacoRepository tacoRepo) {
        this.tacoRepo = tacoRepo;
    }

    @GetMapping("/recent")
    public Iterable<Taco> recentTacos() {
        PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
        return tacoRepo.findAll(page).getContent();
    }
}

```

The code snippet shows the `DesignTacoController` class annotated with `@RestController`, `@RequestMapping`, and `@CrossOrigin`. Callouts explain the purpose of each annotation:

- `@RestController`: Handles requests for `/design`
- `@CrossOrigin(origins="*")`: Allows cross-origin requests
- `@GetMapping("/recent")`: Fetches and returns recent taco designs

You may be thinking that this controller's name sounds familiar. In chapter 2 you created a `DesignTacoController` that handled similar types of requests. But where that controller was for the multipage Taco Cloud application, this new `DesignTacoController` is a REST controller, as indicated by the `@RestController` annotation.

The `@RestController` annotation serves two purposes. First, it's a stereotype annotation like `@Controller` and `@Service` that marks a class for discovery by component scanning. But most relevant to the discussion of REST, the `@RestController` annotation tells Spring that all handler methods in the controller should have their return value written directly to the body of the response, rather than being carried in the model to a view for rendering.

Alternatively, you could have annotated `DesignTacoController` with `@Controller`, just like with any Spring MVC controller. But then you'd need to also annotate all of the handler methods with `@ResponseBody` to achieve the same result. Yet another option would be to return a `ResponseEntity` object, which we'll discuss in a moment.

The `@RequestMapping` annotation at the class level works with the `@GetMapping` annotation on the `recentTacos()` method to specify that the `recentTacos()` method

is responsible for handling GET requests for /design/recent (which is exactly what your Angular code needs).

You'll notice that the `@RequestMapping` annotation also sets a `produces` attribute. This specifies that any of the handler methods in `DesignTacoController` will only handle requests if the request's `Accept` header includes "application/json". Not only does this limit your API to only producing JSON results, it also allows for another controller (perhaps the `DesignTacoController` from chapter 2) to handle requests with the same paths, so long as those requests don't require JSON output. Even though this limits your API to being JSON-based (which is fine for your needs), you're welcome to set `produces` to an array of `String` for multiple content types. For example, to allow for XML output, you could add "text/html" to the `produces` attribute:

```
@RequestMapping(path="/design",
    produces={"application/json", "text/xml"})
```

The other thing you may have noticed in listing 6.2 is that the class is annotated with `@CrossOrigin`. Because the Angular portion of the application will be running on a separate host and/or port from the API (at least for now), the web browser will prevent your Angular client from consuming the API. This restriction can be overcome by including CORS (Cross-Origin Resource Sharing) headers in the server responses. Spring makes it easy to apply CORS with the `@CrossOrigin` annotation. As applied here, `@CrossOrigin` allows clients from any domain to consume the API.

The logic within the `recentTacos()` method is fairly straightforward. It constructs a `PageRequest` object that specifies that you only want the first (0th) page of 12 results, sorted in descending order by the taco's creation date. In short, you want a dozen of the most recently created taco designs. The `PageRequest` is passed into the call to the `findAll()` method of `TacoRepository`, and the content of that page of results is returned to the client (which, as you saw in listing 6.1, will be used as model data to display to the user).

Now let's say that you want to offer an endpoint that fetches a single taco by its ID. By using a placeholder variable in the handler method's path and accepting a path variable, you can capture the ID and use it to look up the `Taco` object through the repository:

```
@GetMapping("/{id}")
public Taco tacoById(@PathVariable("id") Long id) {
    Optional<Taco> optTaco = tacoRepo.findById(id);
    if (optTaco.isPresent()) {
        return optTaco.get();
    }
    return null;
}
```

Because the controller's base path is /design, this controller method handles GET requests for /design/{id}, where the `{id}` portion of the path is a placeholder. The

actual value in the request is given to the `id` parameter, which is mapped to the `{id}` placeholder by `@PathVariable`.

Inside of `tacoById()`, the `id` parameter is passed to the repository's `findById()` method to fetch the `Taco`. `findById()` returns an `Optional<Taco>` because there may not be a `Taco` with the given ID. Therefore, you need to determine whether the ID matched a `Taco` or not before returning a value. If it matches, you call `get()` on the `Optional<Taco>` object to return the actual `Taco`.

If the ID doesn't match any known `Tacos`, you return `null`. This, however, is less than ideal. By returning `null`, the client receives a response with an empty body and an HTTP status code of 200 (OK). The client is handed a response it can't use, but the status code indicates everything is fine. A better approach would be to return a response with an HTTP 404 (NOT FOUND) status.

As it's currently written, there's no easy way to return a 404 status code from `tacoById()`. But if you make a few small tweaks, you can set the status code appropriately:

```
@GetMapping("/{id}")
public ResponseEntity<Taco> tacoById(@PathVariable("id") Long id) {
    Optional<Taco> optTaco = tacoRepo.findById(id);
    if (optTaco.isPresent()) {
        return new ResponseEntity<>(optTaco.get(), HttpStatus.OK);
    }
    return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
}
```

Now, instead of returning a `Taco` object, `tacoById()` returns a `ResponseEntity<Taco>`. If the `Taco` is found, you wrap the `Taco` object in a `ResponseEntity` with an HTTP status of OK (which is what the behavior was before). But if the `Taco` isn't found, you wrap a `null` in a `ResponseEntity` along with an HTTP status of NOT FOUND to indicate that the client is trying to fetch a `Taco` that doesn't exist.

You now have the start of a `Taco Cloud API` for your Angular client—or any other kind of client, for that matter. For development testing purposes, you may also want to use command-line utilities like `curl` or `HTTPie` (<https://httpie.org/>) to poke about the API. For example, the following command line shows how you might fetch recently created `Tacos` with `curl`:

```
$ curl localhost:8080/design/recent
```

Or like this if you prefer `HTTPie`:

```
$ http :8080/design/recent
```

But defining an endpoint that returns information is only the start. What if your API needs to receive data from the client? Let's see how you can write controller methods that handle input on the requests.

6.1.2 Sending data to the server

So far your API is able to return a dozen of the most recently created tacos. But how did those tacos get created in the first place?

You haven't deleted any code from chapter 2 yet, so you still have the original `DesignTacoController` that displays a taco design form and handles form submission. That's a great way to get some test data in place to test the API you've created. But if you're going to transform Taco Cloud into a single-page application, you'll need to create Angular components and corresponding endpoints to replace that taco design form from chapter 2.

I've already handled the client code for the taco design form by defining a new Angular component named `DesignComponent` (in a file named `design.component.ts`). As it pertains to handling form submission, `DesignComponent` has an `onSubmit()` method that looks like this:

```
onSubmit() {
  this.httpClient.post(
    'http://localhost:8080/design',
    this.model,
    {
      headers: new HttpHeaders().set('Content-type', 'application/json'),
    }).subscribe(taco => this.cart.addToCart(taco));

  this.router.navigate(['/cart']);
}
```

In the `onSubmit()` method, the `post()` method of `HttpClient` is called instead of `get()`. This means that instead of fetching data from the API, you're sending data to the API. Specifically, you're sending a taco design, which is held in the `model` variable, to the API endpoint at `/design` with an HTTP POST request.

This means that you'll need to write a method in `DesignTacoController` to handle that request and save the design. By adding the following `postTaco()` method to `DesignTacoController`, you enable the controller to do exactly that:

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
  return tacoRepo.save(taco);
}
```

Because `postTaco()` will handle an HTTP POST request, it's annotated with `@PostMapping` instead of `@GetMapping`. You're not specifying a path attribute here, so the `postTaco()` method will handle requests for `/design` as specified in the class-level `@RequestMapping` on `DesignTacoController`.

You do set the `consumes` attribute, however. The `consumes` attribute is to request input what produces is to request output. Here you use `consumes` to say that the method will only handle requests whose `Content-type` matches `application/json`.

The `Taco` parameter to the method is annotated with `@RequestBody` to indicate that the body of the request should be converted to a `Taco` object and bound to the

parameter. This annotation is important—without it, Spring MVC would assume that you want request parameters (either query parameters or form parameters) to be bound to the `Taco` object. But the `@RequestBody` annotation ensures that JSON in the request body is bound to the `Taco` object instead.

Once `postTaco()` has received the `Taco` object, it passes it to the `save()` method on the `TacoRepository`.

You may have also noticed that I've annotated the `postTaco()` method with `@ResponseStatus(HttpStatus.CREATED)`. Under normal circumstances (when no exceptions are thrown), all responses will have an HTTP status code of 200 (OK), indicating that the request was successful. Although an HTTP 200 response is always welcome, it's not always descriptive enough. In the case of a POST request, an HTTP status of 201 (CREATED) is more descriptive. It tells the client that not only was the request successful, but a resource was created as a result. It's always a good idea to use `@ResponseStatus` where appropriate to communicate the most descriptive and accurate HTTP status code to the client.

Although you've used `@PostMapping` to create a new `Taco` resource, POST requests can also be used to update resources. Even so, POST requests are typically used for resource creation and PUT and PATCH requests are used to update resources. Let's see how you can update data using `@PutMapping` and `@PatchMapping`.

6.1.3 *Updating data on the server*

Before you write any controller code for handling HTTP PUT or PATCH commands, you should take a moment to consider the elephant in the room: Why are there two different HTTP methods for updating resources?

Although it's true that PUT is often used to update resource data, it's actually the semantic opposite of GET. Whereas GET requests are for transferring data from the server to the client, PUT requests are for sending data from the client to the server.

In that sense, PUT is really intended to perform a wholesale *replacement* operation rather than an update operation. In contrast, the purpose of HTTP PATCH is to perform a patch or partial update of resource data.

For example, suppose you want to be able to change the address on an order. One way we could achieve this through the REST API is with a PUT request handled like this:

```
@PutMapping("/{orderId}")
public Order putOrder(@RequestBody Order order) {
    return repo.save(order);
}
```

This could work, but it would require that the client submit the complete order data in the PUT request. Semantically, PUT means “put this data at this URL,” essentially replacing any data that's already there. If any of the order's properties are omitted, that property's value would be overwritten with `null`. Even the tacos in the order would need to be set along with the order data or else they'd be removed from the order.

If PUT does a wholesale replacement of the resource data, then how should you handle requests to do just a partial update? That's what HTTP PATCH requests and Spring's @PatchMapping are good for. Here's how you might write a controller method to handle a PATCH request for an order:

```
@PatchMapping(path="/{orderId}", consumes="application/json")
public Order patchOrder(@PathVariable("orderId") Long orderId,
                        @RequestBody Order patch) {

    Order order = repo.findById(orderId).get();
    if (patch.getDeliveryName() != null) {
        order.setDeliveryName(patch.getDeliveryName());
    }
    if (patch.getDeliveryStreet() != null) {
        order.setDeliveryStreet(patch.getDeliveryStreet());
    }
    if (patch.getDeliveryCity() != null) {
        order.setDeliveryCity(patch.getDeliveryCity());
    }
    if (patch.getDeliveryState() != null) {
        order.setDeliveryState(patch.getDeliveryState());
    }
    if (patch.getDeliveryZip() != null) {
        order.setDeliveryZip(patch.getDeliveryState());
    }
    if (patch.getCcNumber() != null) {
        order.setCcNumber(patch.getCcNumber());
    }
    if (patch.getCcExpiration() != null) {
        order.setCcExpiration(patch.getCcExpiration());
    }
    if (patch.getCcCVV() != null) {
        order.setCcCVV(patch.getCcCVV());
    }

    return repo.save(order);
}
```

The first thing to note here is that the `patchOrder()` method is annotated with `@PatchMapping` instead of `@PutMapping`, indicating that it should handle HTTP PATCH requests instead of PUT requests.

But the one thing you've no doubt noticed is that the `patchOrder()` method is a bit more involved than the `putOrder()` method. That's because Spring MVC's mapping annotations, including `@PatchMapping` and `@PutMapping`, only specify what kinds of requests a method should handle. These annotations don't dictate how the request will be handled. Even though PATCH semantically implies a partial update, it's up to you to write code in the handler method that actually performs such an update.

In the case of the `putOrder()` method, you accepted the complete data for an order and saved it, adhering to the semantics of HTTP PUT. But in order for `patchMapping()` to adhere to the semantics of HTTP PATCH, the body of the method

requires more intelligence. Instead of completely replacing the order with the new data sent in, it inspects each field of the incoming Order object and applies any non-null values to the existing order. This approach allows the client to only send the properties that should be changed and enables the server to retain existing data for any properties not specified by the client.

There's more than one way to PATCH

The patching approach applied in the `patchOrder()` method has a couple of limitations:

- If null values are meant to specify no change, how can the client indicate that a field should be set to null?
- There's no way of removing or adding a subset of items from a collection. If the client wants to add or remove an entry from a collection, it must send the complete altered collection.

There's really no hard-and-fast rule about how PATCH requests should be handled or what the incoming data should look like. Rather than sending the actual domain data, a client could send a patch-specific description of the changes to be applied. Of course, the request handler would have to be written to handle patch instructions instead of the domain data.

In both `@PutMapping` and `@PatchMapping`, notice that the request path references the resource that's to be changed. This is the same way paths are handled by `@GetMapping`-annotated methods.

You've now seen how to fetch and post resources with `@GetMapping` and `@PostMapping`. And you've seen two different ways of updating a resource with `@PutMapping` and `@PatchMapping`. All that's left is handling requests to delete a resource.

6.1.4 **Deleting data from the server**

Sometimes data simply isn't needed anymore. In those cases, a client should be able to request that a resource be removed with an HTTP DELETE request.

Spring MVC's `@DeleteMapping` comes in handy for declaring methods that handle DELETE requests. For example, let's say you want your API to allow for an order resource to be deleted. The following controller method should do the trick:

```
@DeleteMapping("/{orderId}")
@ResponseBody(code=HttpStatus.NO_CONTENT)
public void deleteOrder(@PathVariable("orderId") Long orderId) {
    try {
        repo.deleteById(orderId);
    } catch (EmptyResultDataAccessException e) {}
}
```

By this point, the idea of another mapping annotation should be old hat to you. You've already seen `@GetMapping`, `@PostMapping`, `@PutMapping`, and `@PatchMapping`—each specifying that a method should handle requests for their corresponding HTTP

methods. It will probably come as no surprise to you that `@DeleteMapping` is used to specify that the `deleteOrder()` method is responsible for handling `DELETE` requests for `/orders/{orderId}`.

The code within the method is what does the actual work of deleting an order. In this case, it takes the order ID, provided as a path variable in the URL, and passes it to the repository's `deleteById()` method. If the order exists when that method is called, it will be deleted. If the order doesn't exist, an `EmptyResultDataAccessException` will be thrown.

I've chosen to catch the `EmptyResultDataAccessException` and do nothing with it. My thinking here is that if you try to delete a resource that doesn't exist, the outcome is the same as if it did exist prior to deletion. That is, the resource will be nonexistent. Whether it existed before or not is irrelevant. Alternatively, I could've written `deleteOrder()` to return a `ResponseEntity`, setting the body to null and the HTTP status code to NOT FOUND.

The only other thing to take note of in the `deleteOrder()` method is that it's annotated with `@ResponseStatus` to ensure that the response's HTTP status is 204 (NO CONTENT). There's no need to communicate any resource data back to the client for a resource that no longer exists, so responses to `DELETE` requests typically have no body and therefore should communicate an HTTP status code to let the client know not to expect any content.

Your Taco Cloud API is starting to take shape. The client-side code can now easily consume this API to present ingredients, accept orders, and display recently created tacos. But there's something you can do that will make your API even easier for the client to consume. Let's look at how you can add hypermedia to the Taco Cloud API.

6.2 Enabling hypermedia

The API you've created thus far is fairly basic, but it does work as long as the client that consumes it is aware of the API's URL scheme. For example, a client may be hardcoded to know that it can obtain a list of recently created tacos by issuing a GET request for `/design/recent`. Likewise, it may be hardcoded to know that it can append the ID of any taco in that list to `/design` to get the URL for that particular taco resource.

Using hardcoded URL patterns and string manipulation is common among API client code. But imagine for a moment what would happen if the API's URL scheme were to change. The hardcoded client code would have an obsolete understanding of the API and would thus be broken. Hardcoding API URLs and using string manipulation on them makes the client code brittle.

Hypermedia as the Engine of Application State, or HATEOAS, is a means of creating self-describing APIs wherein resources returned from an API contain links to related resources. This enables clients to navigate an API with minimal understanding of the API's URLs. Instead, it understands *relationships* between the resources served by the API and uses its understanding of those relationships to discover the API's URLs as it traverses those relationships.

For example, suppose a client were to request a list of recently designed tacos. In its raw form, with no hyperlinks, the list of recent tacos would be received in the client with JSON that looks like this (with all but the first taco in the list clipped out for brevity's sake):

```
[
  {
    "id": 4,
    "name": "Veg-Out",
    "createdAt": "2018-01-31T20:15:53.219+0000",
    "ingredients": [
      {"id": "FLTO", "name": "Flour Tortilla", "type": "WRAP"},
      {"id": "COTO", "name": "Corn Tortilla", "type": "WRAP"},
      {"id": "TOMO", "name": "Diced Tomatoes", "type": "VEGGIES"},
      {"id": "LETC", "name": "Lettuce", "type": "VEGGIES"},
      {"id": "SLSA", "name": "Salsa", "type": "SAUCE"}
    ]
  },
  ...
]
```

If the client wished to fetch or perform some other HTTP operation on the taco itself, it would need to know (via hardcoding) that it could append the value of the `id` property to a URL whose path is `/design`. Likewise, if it wanted to perform an HTTP operation on one of the ingredients, it would need to know that it could append the value of the ingredient's `id` property to a URL whose path is `/ingredients`. In either case, it would also need to prefix that path with `http://` or `https://` and the hostname of the API.

In contrast, if the API is enabled with hypermedia, the API will describe its own URLs, relieving the client of needing to be hardcoded with that knowledge. The same list of recently created tacos might look like the next listing if hyperlinks were embedded.

Listing 6.3 A list of taco resources that includes hyperlinks

```
{
  "_embedded": {
    "tacoResourceList": [
      {
        "name": "Veg-Out",
        "createdAt": "2018-01-31T20:15:53.219+0000",
        "ingredients": [
          {
            "name": "Flour Tortilla", "type": "WRAP",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/FLTO" }
            }
          },
          {
            "name": "Corn Tortilla", "type": "WRAP",
            "_links": {

```

```

        "self": { "href": "http://localhost:8080/ingredients/COTO" }
    },
    {
        "name": "Diced Tomatoes", "type": "VEGGIES",
        "_links": {
            "self": { "href": "http://localhost:8080/ingredients/TMTO" }
        }
    },
    {
        "name": "Lettuce", "type": "VEGGIES",
        "_links": {
            "self": { "href": "http://localhost:8080/ingredients/LETC" }
        }
    },
    {
        "name": "Salsa", "type": "SAUCE",
        "_links": {
            "self": { "href": "http://localhost:8080/ingredients/SLSA" }
        }
    }
],
"_links": {
    "self": { "href": "http://localhost:8080/design/4" }
},
},
...
],
},
"_links": {
    "recents": {
        "href": "http://localhost:8080/design/recent"
    }
}
}
```

This particular flavor of HATEOAS is known as HAL (Hypertext Application Language; http://stateless.co/hal_specification.html), a simple and commonly used format for embedding hyperlinks in JSON responses.

Although this list isn't as succinct as before, it does provide some useful information. Each element in this new list of tacos includes a property named `_links` that contains hyperlinks for the client to navigate the API. In this example, both tacos and ingredients each have `self` links to reference those resources, and the entire list has a `recents` link that references itself.

Should a client application need to perform an HTTP request against a taco in the list, it doesn't need to be developed with any knowledge of what the taco resource's URL would look like. Instead, it knows to ask for the `self` link, which maps to <http://localhost:8080/design/4>. If the client wants to deal with a particular ingredient, it only needs to follow the `self` link for that ingredient.

The Spring HATEOAS project brings hyperlink support to Spring. It offers a set of classes and resource assemblers that can be used to add links to resources before returning them from a Spring MVC controller.

To enable hypermedia in the Taco Cloud API, you'll need to add the Spring HATEOAS starter dependency to the build:

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

This starter not only adds Spring HATEOAS to the project's classpath, but also provides for autoconfiguration to enable Spring HATEOAS. All you need to do is rework your controllers to return resource types instead of domain types.

You'll start by adding hypermedia links to the list of recent tacos returned by a GET request to /design/recent.

6.2.1 Adding hyperlinks

Spring HATEOAS provides two primary types that represent hyperlinked resources: Resource and Resources. The Resource type represents a single resource, whereas Resources is a collection of resources. Both types are capable of carrying links to other resources. When returned from a Spring MVC REST controller method, the links they carry will be included in the JSON (or XML) received by the client.

To add hyperlinks to the list of recently created tacos, you'll need to revisit the recentTacos() method shown in listing 6.2. The original implementation returned a List<Taco>, which was fine at the time, but you're going to need it to return a Resources object instead. The following listing shows a new implementation of recentTacos() that includes the first steps toward enabling hyperlinks in the recent tacos list.

Listing 6.4 Adding hyperlinks to resources

```
@GetMapping("/recent")
public Resources<Resource<Taco>> recentTacos() {
    PageRequest page = PageRequest.of(
        0, 12, Sort.by("createdAt").descending());
    List<Taco> tacos = tacoRepo.findAll(page).getContent();
    Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);

    recentResources.add(
        new Link("http://localhost:8080/design/recent", "recents"));
    return recentResources;
}
```

In this new version of recentTacos(), you no longer return the list of tacos directly. Instead, you use Resources.wrap() to wrap the list of tacos as an instance of

Resources<Resource<Taco>>, which is ultimately returned from the method. But before returning the Resources object, you add a link whose relationship name is `recents` and whose URL is <http://localhost:8080/design/recent>. As a consequence, the following snippet of JSON is included in the resource returned from the API request:

```
_links": {  
    "recents": {  
        "href": "http://localhost:8080/design/recent"  
    }  
}
```

This is a good start, but you've still got some work to do. At this point, the only link you've added is to the entire list; no links are added to the taco resources themselves or to the ingredients of each taco. You'll add those soon. But first, let's address the hardcoded URL that you've given for the `recents` link.

Hardcoding a URL like this is a really bad idea. Unless your Taco Cloud ambitions are limited to only ever running the application on your own development machines, you need a way to not hardcode a URL with `localhost:8080` in it. Fortunately, Spring HATEOAS provides help in the form of link builders.

The most useful of the Spring HATEOAS link builders is `ControllerLinkBuilder`. This link builder is smart enough to know what the hostname is without you having to hardcode it. And it provides a handy fluent API to help you build links relative to the base URL of any controller.

Using `ControllerLinkBuilder`, you can rewrite the hardcoded `Link` creation in `recentTacos()` with the following lines:

```
Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);  
recentResources.add(  
    ControllerLinkBuilder.linkTo(DesignTacoController.class)  
        .slash("recent")  
        .withRel("recents"));
```

Not only do you no longer need to hardcode the hostname, you also don't have to specify the `/design` path. Instead, you ask for a link to `DesignTacoController`, whose base path is `/design`. `ControllerLinkBuilder` uses the controller's base path as the foundation of the `Link` object you're creating.

What's next is a call to one of my favorite methods in any Spring project: `slash()`. I love this method because it so succinctly describes exactly what it's going to do. It quite literally appends a slash (/) and the given value to the URL. As a result, the URL's path is `/design/recent`.

Finally, you specify a relation name for the `Link`. In this example, the relation is named `recents`.

Although I'm a big fan of the `slash()` method, `ControllerLinkBuilder` has another method that can help eliminate any hardcoded associated with link URLs. Instead of calling `slash()`, you can call `linkTo()` by giving it a method on the controller to have `ControllerLinkBuilder` derive the base URL from both the controller's

base path and the method's mapped path. The following code uses the `linkTo()` method this way:

```
Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);
recentResources.add(
    linkTo(methodOn(DesignTacoController.class).recentTacos())
    .withRel("recents"));
```

Here I've decided to statically include the `linkTo()` and `methodOn()` methods (both from `ControllerLinkBuilder`) to keep the code easier to read. The `methodOn()` method takes the controller class and lets you make a call to the `recentTacos()` method, which is intercepted by `ControllerLinkBuilder` and used to determine not only the controller's base path, but also the path mapped to `recentTacos()`. Now the entire URL is derived from the controller's mappings, and absolutely no portion is hardcoded. Sweet!

6.2.2 *Creating resource assemblers*

Now you need to add links to the taco resource contained within the list. One option is to loop through each of the `Resource<Taco>` elements carried in the `Resources` object, adding a `Link` to each individually. But that's a bit tedious and you'd need to repeat that looping code in the API wherever you return a list of taco resources.

We need a different tactic.

Rather than let `Resources.wrap()` create a `Resource` object for each taco in the list, you're going to define a utility class that converts `Taco` objects to a new `TacoResource` object. The `TacoResource` object will look a lot like a `Taco`, but it will also be able to carry links. The next listing shows what a `TacoResource` might look like.

Listing 6.5 A taco resource carrying domain data and a list of hyperlinks

```
package tacos.web.api;
import java.util.Date;
import java.util.List;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Ingredient;
import tacos.Taco;

public class TacoResource extends ResourceSupport {

    @Getter
    private final String name;

    @Getter
    private final Date createdAt;

    @Getter
    private final List<Ingredient> ingredients;

    public TacoResource(Taco taco) {
        this.name = taco.getName();
```

```
        this.createdAt = taco.getCreatedAt();
        this.ingredients = taco.getIngredients();
    }

}
```

In a lot of ways, TacoResource isn't that different from the Taco domain type. They both have name, createdAt, and ingredients properties. But TacoResource extends ResourceSupport to inherit a list of Link object and methods to manage the list of links.

What's more, TacoResource doesn't include the id property from Taco. That's because there's no need to expose any database-specific IDs in the API. The resource's self link will serve as the identifier for the resource from the perspective of an API client.

NOTE Domains and resources: separate or the same? Some Spring developers may choose to combine their domain and resource types into a single type by having their domain types extend ResourceSupport. There's no right or wrong answer as to which is the correct way. I chose to create a separate resource type so that Taco isn't unnecessarily cluttered with resource links for use cases where links aren't needed. Also, by creating a separate resource type, I was able to easily leave the id property out so that it won't be exposed in the API.

TacoResource has a single constructor that accepts a Taco and copies the pertinent properties from the Taco to its own properties. This makes it easy to convert a single Taco object to a TacoResource. But if you stop there, you'd still need looping to convert a list of Taco objects to a Resources<TacoResource>.

To aid in converting Taco objects to TacoResource objects, you're also going to create a resource assembler. The following listing is what you'll need.

Listing 6.6 A resource assembler that assembles taco resources

```
package tacos.web.api;

import org.springframework.hateoas.mvc.ResourceAssemblerSupport;

import tacos.Taco;

public class TacoResourceAssembler
    extends ResourceAssemblerSupport<Taco, TacoResource> {

    public TacoResourceAssembler() {
        super(DesignTacoController.class, TacoResource.class);
    }

    @Override
    protected TacoResource instantiateResource(Taco taco) {
        return new TacoResource(taco);
    }
}
```

```

@Override
public TacoResource toResource(Taco taco) {
    return createResourceWithId(taco.getId(), taco);
}
}

```

TacoResourceAssembler has a default constructor that informs the superclass (`ResourceAssemblerSupport`) that it will be using `DesignTacoController` to determine the base path for any URLs in links it creates when creating a `TacoResource`.

The `instantiateResource()` method is overridden to instantiate a `TacoResource` given a `Taco`. This method would be optional if `TacoResource` had a default constructor. In this case, however, `TacoResource` requires construction with a `Taco`, so you're required to override it.

Finally, the `toResource()` method is the only method that's strictly mandatory when extending `ResourceAssemblerSupport`. Here you're telling it to create a `TacoResource` object from a `Taco`, and to automatically give it a `self` link with the URL being derived from the `Taco` object's `id` property.

On the surface, `toResource()` appears to have a similar purpose to `instantiateResource()`, but they serve slightly different purposes. Whereas `instantiateResource()` is intended to only instantiate a `Resource` object, `toResource()` is intended not only to create the `Resource` object, but also to populate it with links. Under the covers, `toResource()` will call `instantiateResource()`.

Now tweak the `recentTacos()` method to make use of `TacoResourceAssembler`:

```

@GetMapping("/recent")
public Resources<TacoResource> recentTacos() {
    PageRequest page = PageRequest.of(
        0, 12, Sort.by("createdAt").descending());
    List<Taco> tacos = tacoRepo.findAll(page).getContent();
    List<TacoResource> tacoResources =
        new TacoResourceAssembler().toResources(tacos);
    Resources<TacoResource> recentResources =
        new Resources<TacoResource>(tacoResources);
    recentResources.add(
        linkTo(methodOn(DesignTacoController.class).recentTacos())
            .withRel("recents"));
    return recentResources;
}

```

Rather than return a `Resources<Resource<Taco>>`, `recentTacos()` now returns a `Resources<TacoResource>` to take advantage of your new `TacoResource` type. After fetching the `tacos` from the repository, you pass the list of `Taco` objects to the `toResources()` method on a `TacoResourceAssembler`. This handy method cycles through all of the `Taco` objects, calling the `toResource()` method that you overrode in `TacoResourceAssembler` to create a list of `TacoResource` objects.

With that `TacoResource` list, you next create a `Resources<TacoResource>` object and then populate it with the `recents` links as in the prior version of `recentTacos()`.

At this point, a GET request to /design/recent will produce a list of tacos, each with a self link and a recent link on the list itself. But the ingredients will still be without a link. To address that, you'll create a new resource assembler for ingredients:

```
package tacos.web.api;
import org.springframework.hateoas.mvc.ResourceAssemblerSupport;
import tacos.Ingredient;

class IngredientResourceAssembler extends
    ResourceAssemblerSupport<Ingredient, IngredientResource> {

    public IngredientResourceAssembler() {
        super(IngredientController2.class, IngredientResource.class);
    }

    @Override
    public IngredientResource toResource(Ingredient ingredient) {
        return createResourceWithId(ingredient.getId(), ingredient);
    }

    @Override
    protected IngredientResource instantiateResource(
        Ingredient ingredient) {
        return new IngredientResource(ingredient);
    }

}
```

As you can see, IngredientResourceAssembler is much like TacoResourceAssembler, but it works with Ingredient and IngredientResource objects instead of Taco and TacoResource objects.

Speaking of IngredientResource, it looks like this:

```
package tacos.web.api;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Ingredient;
import tacos.Ingredient.Type;

public class IngredientResource extends ResourceSupport {

    @Getter
    private String name;

    @Getter
    private Type type;

    public IngredientResource(Ingredient ingredient) {
        this.name = ingredient.getName();
        this.type = ingredient.getType();
    }

}
```

As with TacoResource, IngredientResource extends ResourceSupport and copies pertinent properties from the domain type into its own set of properties (leaving out the id property).

All that's left is to make a slight change to TacoResource so that it carries IngredientResource objects instead of Ingredient objects:

```
package tacos.web.api;
import java.util.Date;
import java.util.List;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Taco;

public class TacoResource extends ResourceSupport {

    private static final IngredientResourceAssembler
        ingredientAssembler = new IngredientResourceAssembler();

    @Getter
    private final String name;

    @Getter
    private final Date createdAt;

    @Getter
    private final List<IngredientResource> ingredients;

    public TacoResource(Taco taco) {
        this.name = taco.getName();
        this.createdAt = taco.getCreatedAt();
        this.ingredients =
            ingredientAssembler.toResources(taco.getIngredients());
    }

}
```

This new version of TacoResource creates a static final instance of IngredientResourceAssembler and uses its toResource() method to convert a given Taco object's list of Ingredient into a list of IngredientResource.

Your recent tacos list is now completely outfitted with hyperlinks, not only for itself (the recent link), but also for all of its taco entries and the ingredients of those tacos. The response should look a lot like the JSON in listing 6.3.

You could stop here and move on to the next subject. But first I'll address something that's been bugging me about listing 6.3.

6.2.3 Naming embedded relationships

If you take a closer look at listing 6.3, you'll notice that the top-level elements look like this:

```
{  
    "_embedded": {  
        "tacoResourceList": [  
            ...  
        ]  
    }  
}
```

Most notably, let me draw your attention to the name `tacoResourceList` under `embedded`. That name was derived from the fact that the `Resources` object was created from a `List<TacoResource>`. Not that it's likely, but if you were to refactor the name of the `TacoResource` class to something else, the field name in the resulting JSON would change to match it. This would likely break any clients coded to count on that name.

The `@Relation` annotation can help break the coupling between the JSON field name and the resource type class names as defined in Java. By annotating `TacoResource` with `@Relation`, you can specify how Spring HATEOAS should name the field in the resulting JSON:

```
@Relation(value="taco", collectionRelation="tacos")  
public class TacoResource extends ResourceSupport {  
    ...  
}
```

Here you've specified that when a list of `TacoResource` objects is used in a `Resources` object, it should be named `tacos`. And although you're not making use of it in our API, a single `TacoResource` object should be referred to in JSON as `taco`.

As a result, the JSON returned from `/design/recent` will now look like this (no matter what refactoring you may or may not perform on `TacoResource`):

```
{  
    "_embedded": {  
        "tacos": [  
            ...  
        ]  
    }  
}
```

Spring HATEOAS makes adding links to your API rather straightforward and simple. Nonetheless, it did add several lines of code that you wouldn't otherwise need. Because of that, some developers may choose to not bother with HATEOAS in their APIs, even if it means that the client code is subject to breakage if the API's URL scheme changes. I encourage you to take HATEOAS seriously and not to take the lazy way out by not adding hyperlinks in your resources.

But if you insist on being lazy, then maybe there's a win-win scenario for you if you're using Spring Data for your repositories. Let's see how Spring Data REST can help you automatically create APIs based on the data repositories you created with Spring Data in chapter 3.

6.3 **Enabling data-backed services**

As you saw in chapter 3, Spring Data performs a special kind of magic by automatically creating repository implementations based on interfaces you define in your code. But Spring Data has another trick up its sleeve that can help you define APIs for your application.

Spring Data REST is another member of the Spring Data family that automatically creates REST APIs for repositories created by Spring Data. By doing little more than adding Spring Data REST to your build, you get an API with operations for each repository interface you've defined.

To start using Spring Data REST, you add the following dependency to your build:

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

Believe it or not, that's all that's required to expose a REST API in a project that's already using Spring Data for automatic repositories. By simply having the Spring Data REST starter in the build, the application gets auto-configuration that enables automatic creation of a REST API for any repositories that were created by Spring Data (including Spring Data JPA, Spring Data Mongo, and so on).

The REST endpoints that Spring Data REST creates are at least as good as (and possibly even better than) the ones you've created yourself. So at this point, feel free to do a little demolition work and remove any `@RestController`-annotated classes you've created up to this point before moving on.

To try out the endpoints provided by Spring Data REST, you can fire up the application and start poking at some of the URLs. Based on the set of repositories you've already defined for Taco Cloud, you should be able to perform GET requests for tacos, ingredients, orders, and users.

For example, you can get a list of all ingredients by making a GET request for `/ingredients`. Using curl, you might get something that looks like this (abridged to only show the first ingredient):

```
$ curl localhost:8080/ingredients
{
  "_embedded" : {
    "ingredients" : [ {
      "name" : "Flour Tortilla",
      "type" : "WRAP",
      "_links" : {
        "self" : {
```

```
        "href" : "http://localhost:8080/ingredients/FLTO"
    },
    "ingredient" : {
        "href" : "http://localhost:8080/ingredients/FLTO"
    }
}
},
...
]
},
"_links" : {
    "self" : {
        "href" : "http://localhost:8080/ingredients"
    },
    "profile" : {
        "href" : "http://localhost:8080/profile/ingredients"
    }
}
```

Wow! By doing nothing more than adding a dependency to your build, you're not only getting an endpoint for ingredients, but the resources that come back also contain hyperlinks! Pretending to be a client of this API, you can also use curl to follow the self link for the flour tortilla entry:

```
$ curl http://localhost:8080/ingredients/FLTO
{
  "name" : "Flour Tortilla",
  "type" : "WRAP",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    },
    "ingredient" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    }
  }
}
```

To avoid getting too distracted, we won't waste much more time in this book digging into each and every endpoint and option that Spring Data REST has created. But you should know that it also supports `POST`, `PUT`, and `DELETE` methods for the endpoints it creates. That's right: you can `POST` to `/ingredients` to create a new ingredient and `DELETE` `/ingredients/FLTO` to remove flour tortillas from the menu.

One thing you might want to do is set a base path for the API so that its endpoints are distinct and don't collide with any controllers you write. (In fact, if you don't remove the `IngredientsController` you created earlier, it will interfere with the `/ingredients` endpoint provided by Spring Data REST.) To adjust the base path for the API, set the `spring.data.rest.base-path` property:

```
spring:
  data:
    rest:
      base-path: /api
```

This sets the base path for Spring Data REST endpoints to /api. Consequently, the ingredients endpoint is now /api/ingredients. Now give this new base path a spin by requesting a list of tacos:

```
$ curl http://localhost:8080/api/tacos
{
  "timestamp": "2018-02-11T16:22:12.381+0000",
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/api/tacos"
}
```

Oh dear! That didn't work quite as expected. You have an `Ingredient` entity and an `IngredientRepository` interface, which Spring Data REST exposed with an /api/ingredients endpoint. So if you have a `Taco` entity and a `TacoRepository` interface, why doesn't Spring Data REST give you an /api/tacos endpoint?

6.3.1 **Adjusting resource paths and relation names**

Actually, Spring Data REST *does* give you an endpoint for working with tacos. But as clever as Spring Data REST can be, it shows itself to be a tiny bit less awesome in how it exposes the tacos endpoint.

When creating endpoints for Spring Data repositories, Spring Data REST tries to pluralize the associated entity class. For the `Ingredient` entity, the endpoint is /ingredients. For the `Order` and `User` entities it's /orders and /users. So far, so good.

But sometimes, such as with “taco”, it trips up on a word and the pluralized version isn't quite right. As it turns out, Spring Data REST pluralized “taco” as “tacoes”, so to make a request for tacos, you must play along and request /api/tacoes:

```
% curl localhost:8080/api/tacoes
{
  "_embedded" : {
    "tacoes" : [ {
      "name" : "Carnivore",
      "createdAt" : "2018-02-11T17:01:32.999+0000",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api/tacoes/2"
        },
        "taco" : {
          "href" : "http://localhost:8080/api/tacoes/2"
        },
        "ingredients" : {
          "href" : "http://localhost:8080/api/tacoes/2/ingredients"
        }
      }
    }
  }
}
```

```

        }
    ],
},
"page" : {
    "size" : 20,
    "totalElements" : 3,
    "totalPages" : 1,
    "number" : 0
}
}
}
```

You may be wondering how I knew that “taco” would be mispluralized as “tacos”. As it turns out, Spring Data REST also exposes a home resource that has links for all exposed endpoints. Just make a GET request to the API base path to get the goods:

```
$ curl localhost:8080/api
{
  "_links" : {
    "orders" : {
      "href" : "http://localhost:8080/api/orders"
    },
    "ingredients" : {
      "href" : "http://localhost:8080/api/ingredients"
    },
    "tacos" : {
      "href" : "http://localhost:8080/api/tacos{?page,size,sort}",
      "templated" : true
    },
    "users" : {
      "href" : "http://localhost:8080/api/users"
    },
    "profile" : {
      "href" : "http://localhost:8080/api/profile"
    }
  }
}
```

As you can see, the home resource shows the links for all of your entities. Everything looks good, except for the tacos link, where both the relation name and the URL have the odd pluralization of “taco”.

The good news is that you don’t have to accept this little quirk of Spring Data REST. By adding a simple annotation to the Taco class, you can tweak both the relation name and that path:

```
@Data
@Entity
@RestResource(rel="tacos", path="tacos")
public class Taco {
    ...
}
```

The `@RestResource` annotation lets you give the entity any relation name and path you want. In this case, you’re setting them both to “tacos”. Now when you request the home resource, you see the `tacos` link with correct pluralization:

```
"tacos" : {
    "href" : "http://localhost:8080/api/tacos{?page,size,sort}",
    "templated" : true
},
```

This also sorts out the path for the endpoint so that you can issue requests against `/api/tacos` to work with taco resources.

Speaking of sorting things out, let’s look at how you can sort the results from Spring Data REST endpoints.

6.3.2 *Paging and sorting*

You may have noticed that the links in the home resource all offer optional `page`, `size`, and `sort` parameters. By default, requests to a collection resource such as `/api/tacos` will return up to 20 items per page from the first page. But you can adjust the page size and the page displayed by specifying the `page` and `size` parameters in your request.

For example, to request the first page of tacos where the page size is 5, you can issue the following GET request (using curl):

```
$ curl "localhost:8080/api/tacos?size=5"
```

Assuming that there are more than five tacos to be seen, you can request the second page of tacos by adding the `page` parameter:

```
$ curl "localhost:8080/api/tacos?size=5&page=1"
```

Notice that the `page` parameter is zero-based, which means that asking for page 1 is actually asking for the second page. (You’ll also note that many command-line shells trip up over the ampersand in the request, which is why I quoted the whole URL in the preceding curl command.)

You could use string manipulation to add those parameters to the URL, but HATEOAS comes to the rescue by offering links for the first, last, next, and previous pages in the response:

```
"_links" : {
    "first" : {
        "href" : "http://localhost:8080/api/tacos?page=0&size=5"
    },
    "self" : {
        "href" : "http://localhost:8080/api/tacos"
    },
    "next" : {
        "href" : "http://localhost:8080/api/tacos?page=1&size=5"
    },
```

```

    "last" : {
      "href" : "http://localhost:8080/api/tacos?page=2&size=5"
    },
    "profile" : {
      "href" : "http://localhost:8080/api/profile/tacos"
    },
    "recents" : {
      "href" : "http://localhost:8080/api/tacos/recent"
    }
  }
}

```

With these links, a client of the API need not keep track of what page it's on and concatenate the parameters to the URL. Instead, it must simply know to look for one of these page navigation links by its name and follow it.

The `sort` parameter lets you sort the resulting list by any property of the entity. For example, you need a way to fetch the 12 most recently created tacos for the UI to display. You can do that by specifying the following mix of paging and sorting parameters:

```
$ curl "localhost:8080/api/tacos?sort=createdAt,desc&page=0&size=12"
```

Here the `sort` parameter specifies that you should sort by the `createdAt` property and that it should be sorted in descending order (so that the newest tacos are first). The `page` and `size` parameters specify that you should see the first page of 12 tacos.

This is precisely what the UI needs in order to show the most recently created tacos. It's approximately the same as the `/design/recent` endpoint you defined in `DesignTacoController` earlier in this chapter.

There's a small problem, though. The UI code will need to be hardcoded to request the list of tacos with those parameters. Sure, it will work. But you're adding some brittleness to the client by making it too knowledgeable regarding how to construct an API request. It would be great if the client could look up the URL from a list of links. And it would be even more awesome if the URL were more succinct, like the `/design/recent` endpoint you had before.

6.3.3 Adding custom endpoints

Spring Data REST is great at creating endpoints for performing CRUD operations against Spring Data repositories. But sometimes you need to break away from the default CRUD API and create an endpoint that gets to the core of the problem.

There's absolutely nothing stopping you from implementing any endpoint you want in a `@RestController`-annotated bean to supplement what Spring Data REST automatically generates. In fact, you could resurrect the `DesignTacoController` from earlier in the chapter, and it would still work alongside the endpoints provided by Spring Data REST.

But when you write your own API controllers, their endpoints seem somewhat detached from the Spring Data REST endpoints in a couple of ways:

- Your own controller endpoints aren't mapped under Spring Data REST's base path. You could force their mappings to be prefixed with whatever base path you want, including the Spring Data REST base path, but if the base path were to change, you'd need to edit the controller's mappings to match.
- Any endpoints you define in your own controllers won't be automatically included as hyperlinks in the resources returned by Spring Data REST endpoints. This means that clients won't be able to discover your custom endpoints with a relation name.

Let's address the concern about the base path first. Spring Data REST includes `@RepositoryRestController`, a new annotation for annotating controller classes whose mappings should assume a base path that's the same as the one configured for Spring Data REST endpoints. Put simply, all mappings in a `@RepositoryRestController`-annotated controller will have their path prefixed with the value of the `spring.data.rest.base-path` property (which you've configured as `/api`).

Rather than resurrect the `DesignTacoController`, which had several handler methods you won't need, you'll create a new controller that only contains the `recentTacos()` method. `RecentTacosController` in the next listing is annotated with `@RepositoryRestController` to adopt Spring Data REST's base path for its request mappings.

Listing 6.7 Applying Spring Data REST's base path to a controller

```
package tacos.web.api;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;
import java.util.List;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.data.rest.webmvc.RepositoryRestController;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import tacos.Taco;
import tacos.data.TacoRepository;

@RepositoryRestController
public class RecentTacosController {

    private TacoRepository tacoRepo;

    public RecentTacosController(TacoRepository tacoRepo) {
        this.tacoRepo = tacoRepo;
    }

    @GetMapping(path="/tacos/recent", produces="application/hal+json")
    public ResponseEntity<Resources<TacoResource>> recentTacos() {
```

```
PageRequest page = PageRequest.of(
    0, 12, Sort.by("createdAt").descending());
List<Taco> tacos = tacoRepo.findAll(page).getContent();

List<TacoResource> tacoResources =
    new TacoResourceAssembler().toResources(tacos);
Resources<TacoResource> recentResources =
    new Resources<TacoResource>(tacoResources);
recentResources.add(
    linkTo(methodOn(RecentTacosController.class).recentTacos())
        .withRel("recents"));
return new ResponseEntity<>(recentResources, HttpStatus.OK);
}

}
```

Even though `@GetMapping` is mapped to the path `/tacos/recent`, the `@RepositoryRestController` annotation at the class level will ensure that it will be prefixed with Spring Data REST's base path. As you've configured it, the `recentTacos()` method will handle GET requests for `/api/tacos/recent`.

One important thing to notice is that although `@RepositoryRestController` is named similarly to `@RestController`, it doesn't carry the same semantics as `@RestController`. Specifically, it doesn't ensure that values returned from handler methods are automatically written to the body of the response. Therefore you need to either annotate the method with `@ResponseBody` or return a `ResponseEntity` that wraps the response data. Here you chose to return a `ResponseEntity`.

With `RecentTacosController` in play, requests for `/api/tacos/recent` will return up to 15 of the most recently created tacos, without the need for paging and sorting parameters in the URL. But it still doesn't appear in the hyperlinks list when requesting `/api/tacos`. Let's fix that.

6.3.4 Adding custom hyperlinks to Spring Data endpoints

If the recent tacos endpoint isn't among the hyperlinks returned from `/api/tacos`, how will a client know how to fetch the most recent tacos? It'll either have to guess or use the paging and sorting parameters. Either way, it'll be hardcoded in the client code, which isn't ideal.

By declaring a resource processor bean, however, you can add links to the list of links that Spring Data REST automatically includes. Spring Data HATEOAS offers `ResourceProcessor`, an interface for manipulating resources before they're returned through the API. For your purposes, you need an implementation of `ResourceProcessor` that adds a `recents` link to any resource of type `PagedResources<Resource<Taco>>` (the type returned for the `/api/tacos` endpoint). The next listing shows a bean declaration method that defines such a `ResourceProcessor`.

Listing 6.8 Adding custom links to a Spring Data REST endpoint

```
@Bean
public ResourceProcessor<PagedResources<Resource<Taco>>>
tacoProcessor(EntityLinks links) {
    return new ResourceProcessor<PagedResources<Resource<Taco>>>() {
        @Override
        public PagedResources<Resource<Taco>> process(
            PagedResources<Resource<Taco>> resource) {
            resource.add(
                links.linkFor(Taco.class)
                    .slash("recent")
                    .withRel("recents"));
            return resource;
        }
    };
}
```

The `ResourceProcessor` shown in listing 6.8 is defined as an anonymous inner class and declared as a bean to be created in the Spring application context. Spring HATEOAS will discover this bean (as well as any other beans of type `ResourceProcessor`) automatically and will apply them to the appropriate resources. In this case, if a `PagedResources<Resource<Taco>>` is returned from a controller, it will receive a link for the most recently created tacos. This includes the response for requests for `/api/tacos`.

Summary

- REST endpoints can be created with Spring MVC, with controllers that follow the same programming model as browser-targeted controllers.
- Controller handler methods can either be annotated with `@ResponseBody` or return `ResponseEntity` objects to bypass the model and view and write data directly to the response body.
- The `@RestController` annotation simplifies REST controllers, eliminating the need to use `@ResponseBody` on handler methods.
- Spring HATEOAS enables hyperlinking of resources returned from Spring MVC controllers.
- Spring Data repositories can automatically be exposed as REST APIs using Spring Data REST.



Consuming REST services

This chapter covers

- Using RestTemplate to consume REST APIs
- Navigating hypermedia APIs with Traverson

Have you ever gone to a movie and, as the movie starts, discovered that you were the only person in the theater? It certainly is a wonderful experience to have what is essentially a private viewing of a movie. You can pick whatever seat you want, talk back to the characters onscreen, and maybe even open your phone and tweet about it without anyone getting angry for disrupting their movie-watching experience. And the best part is that nobody else is there ruining the movie for you, either!

This hasn't happened to me often. But when it has, I have wondered what would have happened if I hadn't shown up. Would they still have shown the film? Would the hero still have saved the day? Would the theater staff still have cleaned the theater after the movie was over?

A movie without an audience is kind of like an API without a client. It's ready to accept and provide data, but if the API is never invoked, is it really an API? Like Schrödinger's cat, we can't know if the API is active or returning HTTP 404 responses until we issue a request to it.

In the previous chapter, we focused on defining REST endpoints that can be consumed by some client external to your application. Although the driving force

for developing such an API was a single-page Angular application that served as the Taco Cloud website, the reality is that the client could be any application, in any language—even another Java application.

It's not uncommon for Spring applications to both provide an API and make requests to another application's API. In fact, this is becoming prevalent in the world of microservices. Therefore, it's worthwhile to spend a moment looking at how to use Spring to interact with REST APIs.

A Spring application can consume a REST API with

- `RestTemplate`—A straightforward, synchronous REST client provided by the core Spring Framework.
- `Traverson`—A hyperlink-aware, synchronous REST client provided by Spring HATEOAS. Inspired from a JavaScript library of the same name.
- `WebClient`—A reactive, asynchronous REST client introduced in Spring 5.

I'll defer discussion of `WebClient` until we cover Spring's reactive web framework in chapter 11. For now, we'll focus on the other two REST clients, starting with `RestTemplate`.

7.1 Consuming REST endpoints with `RestTemplate`

There's a lot that goes into interacting with a REST resource from the client's perspective—mostly tedium and boilerplate. Working with low-level HTTP libraries, the client needs to create a client instance and a request object, execute the request, interpret the response, map the response to domain objects, and handle any exceptions that may be thrown along the way. And all of this boilerplate is repeated, regardless of what HTTP request is sent.

To avoid such boilerplate code, Spring provides `RestTemplate`. Just as `JDBC-Template` handles the ugly parts of working with JDBC, `RestTemplate` frees you from dealing with the tedium of consuming REST resources.

`RestTemplate` provides 41 methods for interacting with REST resources. Rather than examine all of the methods that it offers, it's easier to consider only a dozen unique operations, each overloaded to equal the complete set of 41 methods. The 12 operations are described in table 7.1.

Table 7.1 `RestTemplate` defines 12 unique operations, each of which is overloaded, providing a total of 41 methods.

Method	Description
<code>delete(...)</code>	Performs an HTTP DELETE request on a resource at a specified URL
<code>exchange(...)</code>	Executes a specified HTTP method against a URL, returning a <code>ResponseEntity</code> containing an object mapped from the response body
<code>execute(...)</code>	Executes a specified HTTP method against a URL, returning an object mapped from the response body

Table 7.1 RestTemplate defines 12 unique operations, each of which is overloaded, providing a total of 41 methods. (continued)

Method	Description
getForEntity(...)	Sends an HTTP GET request, returning a ResponseEntity containing an object mapped from the response body
getForObject (...)	Sends an HTTP GET request, returning an object mapped from a response body
headForHeaders (...)	Sends an HTTP HEAD request, returning the HTTP headers for the specified resource URL
optionsForAllow(...)	Sends an HTTP OPTIONS request, returning the Allow header for the specified URL
patchForObject (...)	Sends an HTTP PATCH request, returning the resulting object mapped from the response body
postForEntity (...)	POSTs data to a URL, returning a ResponseEntity containing an object mapped from the response body
postForLocation (...)	POSTs data to a URL, returning the URL of the newly created resource
postForObject (...)	POSTs data to a URL, returning an object mapped from the response body
put (...)	PUTs resource data to the specified URL

With the exception of TRACE, RestTemplate has at least one method for each of the standard HTTP methods. In addition, execute() and exchange() provide lower-level, general-purpose methods for sending requests with any HTTP method.

Most of the methods in table 7.1 are overloaded into three method forms:

- One accepts a String URL specification with URL parameters specified in a variable argument list.
- One accepts a String URL specification with URL parameters specified in a Map<String, String>.
- One accepts a java.net.URI as the URL specification, with no support for parameterized URLs.

Once you get to know the 12 operations provided by RestTemplate and how each of the variant forms works, you'll be well on your way to writing resource-consuming REST clients.

To use RestTemplate, you'll either need to create an instance at the point you need it

```
RestTemplate rest = new RestTemplate();
```

or you can declare it as a bean and inject it where you need it:

```
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

Let's survey `RestTemplate`'s operations by looking at those that support the four primary HTTP methods: GET, PUT, DELETE, and POST. We'll start with `getForObject()` and `getForEntity()`—the GET methods.

7.1.1 GETting resources

Suppose that you want to fetch an ingredient from the Taco Cloud API. Assuming that the API isn't HATEOAS-enabled, you can use `getForObject()` to fetch the ingredient. For example, the following code uses `RestTemplate` to fetch an `Ingredient` object by its ID:

```
public Ingredient getIngredientById(String ingredientId) {
    return rest.getForObject("http://localhost:8080/ingredients/{id}",
        Ingredient.class, ingredientId);
}
```

Here you're using the `getForObject()` variant that accepts a `String` URL and uses a variable list for URL variables. The `ingredientId` parameter passed into `getForObject()` is used to fill in the `{id}` placeholder in the given URL. Although there's only one URL variable in this example, it's important to know that the variable parameters are assigned to the placeholders in the order that they're given.

The second parameter to `getForObject()` is the type that the response should be bound to. In this case, the response data (that's likely in JSON format) should be deserialized into an `Ingredient` object that will be returned.

Alternatively, you can use a `Map` to specify the URL variables:

```
public Ingredient getIngredientById(String ingredientId) {
    Map<String, String> urlVariables = new HashMap<>();
    urlVariables.put("id", ingredientId);
    return rest.getForObject("http://localhost:8080/ingredients/{id}",
        Ingredient.class, urlVariables);
}
```

In this case, the value of `ingredientId` is mapped to a key of `id`. When the request is made, the `{id}` placeholder is replaced by the map entry whose key is `id`.

Using a URI parameter is a bit more involved, requiring that you construct a URI object before calling `getForObject()`. Otherwise, it's similar to both of the other variants:

```
public Ingredient getIngredientById(String ingredientId) {
    Map<String, String> urlVariables = new HashMap<>();
    urlVariables.put("id", ingredientId);
    URI url = UriComponentsBuilder
        .fromHttpUrl("http://localhost:8080/ingredients/{id}")
        .build(urlVariables);

    return rest.getForObject(url, Ingredient.class);
}
```

Here the `URI` object is defined from a `String` specification, and its placeholders filled in from entries in a `Map`, much like the previous variant of `getForObject()`. The `getForObject()` method is a no-nonsense way of fetching a resource. But if the client needs more than the payload body, you may want to consider using `getForEntity()`.

`getForEntity()` works in much the same way as `getForObject()`, but instead of returning a domain object that represents the response's payload, it returns a `ResponseEntity` object that wraps that domain object. The `ResponseEntity` gives access to additional response details, such as the response headers.

For example, suppose that in addition to the ingredient data, you want to inspect the `Date` header from the response. With `getForEntity()` that becomes straightforward:

```
public Ingredient getIngredientById(String ingredientId) {
    ResponseEntity<Ingredient> responseEntity =
        rest.getForEntity("http://localhost:8080/ingredients/{id}",
                          Ingredient.class, ingredientId);

    log.info("Fetched time: " +
             responseEntity.getHeaders().getDate());

    return responseEntity.getBody();
}
```

The `getForEntity()` method is overloaded with the same parameters as `getForObject()`, so you can provide the URL variables as a variable list parameter or call `getForEntity()` with a `URI` object.

7.1.2 PUTting resources

For sending HTTP PUT requests, `RestTemplate` offers the `put()` method. All three overloaded variants of `put()` accept an `Object` that is to be serialized and sent to the given URL. As for the URL itself, it can be specified as a `URI` object or as a `String`. And like `getForObject()` and `getForEntity()`, the URL variables can be provided as either a variable argument list or as a `Map`.

Suppose that you want to replace an ingredient resource with the data from a new `Ingredient` object. The following code should do the trick:

```
public void updateIngredient(Ingredient ingredient) {
    rest.put("http://localhost:8080/ingredients/{id}",
             ingredient,
             ingredient.getId());
}
```

Here the URL is given as a `String` and has a placeholder that's substituted by the given `Ingredient` object's `id` property. The data to be sent is the `Ingredient` object itself. The `put()` method returns `void`, so there's nothing you need to do to handle a return value.

7.1.3 **DELETEing resources**

Suppose that Taco Cloud no longer offers an ingredient and wants it completely removed as an option. To make that happen, you can call the `delete()` method from `RestTemplate`:

```
public void deleteIngredient(Ingredient ingredient) {  
    rest.delete("http://localhost:8080/ingredients/{id}",  
                ingredient.getId());  
}
```

In this example, only the URL (specified as a `String`) and a URL variable value are given to `delete()`. But as with the other `RestTemplate` methods, the URL could be specified as a `URI` object or the URL parameters given as a `Map`.

7.1.4 **POSTing resource data**

Now let's say that you add a new ingredient to the Taco Cloud menu. An HTTP `POST` request to the `.../ingredients` endpoint with ingredient data in the request body will make that happen. `RestTemplate` has three ways of sending a `POST` request, each of which has the same overloaded variants for specifying the URL. If you wanted to receive the newly created `Ingredient` resource after the `POST` request, you'd use `postForObject()` like this:

```
public Ingredient createIngredient(Ingredient ingredient) {  
    return rest.postForObject("http://localhost:8080/ingredients",  
                             ingredient,  
                             Ingredient.class);  
}
```

This variant of the `postForObject()` method takes a `String` URL specification, the object to be posted to the server, and the domain type that the response body should be bound to. Although you aren't taking advantage of it in this case, a fourth parameter could be a `Map` of the URL variable value or a variable list of parameters to substitute into the URL.

If your client has more need for the location of the newly created resource, then you can call `postForLocation()` instead:

```
public URI createIngredient(Ingredient ingredient) {  
    return rest.postForLocation("http://localhost:8080/ingredients",  
                               ingredient);  
}
```

Notice that `postForLocation()` works much like `postForObject()` with the exception that it returns a `URI` of the newly created resource instead of the resource object itself. The `URI` returned is derived from the response's `Location` header. In the off chance that you need both the location and response payload, you can call `postForEntity()`:

```

public Ingredient createIngredient(Ingredient ingredient) {
    ResponseEntity<Ingredient> responseEntity =
        rest.postForEntity("http://localhost:8080/ingredients",
                           ingredient,
                           Ingredient.class);

    log.info("New resource created at " +
            responseEntity.getHeaders().getLocation());

    return responseEntity.getBody();
}

```

Although the methods of `RestTemplate` differ in their purpose, they're quite similar in how they're used. This makes it easy to become proficient with `RestTemplate` and use it in your client code.

On the other hand, if the API you're consuming includes hyperlinks in its response, `RestTemplate` isn't as helpful. It's certainly possible to fetch the more detailed resource data with `RestTemplate` and work with the content and links contained therein, but it's not trivial to do so. Rather than struggle while consuming hypermedia APIs with `RestTemplate`, let's turn our attention to a client library that's made for such things—Traverson.

7.2 Navigating REST APIs with Traverson

Traverson comes with Spring Data HATEOAS as the out-of-the-box solution for consuming hypermedia APIs in Spring applications. This Java-based library is inspired by a similar JavaScript library of the same name (<https://github.com/traverser/traverser>).

You might have noticed that Traverson's name kind of sounds like "traverse on", which is a good way to describe how it's used. In this section, you'll consume an API by traversing the API on relation names.

Working with Traverson starts with instantiating a `Traverser` object with an API's base URI:

```

Traverser traverser = new Traverser(
    Uri.create("http://localhost:8080/api"), MediaTypes.HAL_JSON);

```

Here I've pointed `Traverser` to the Taco Cloud's base URL (running locally). This is the only URL you'll need to give to `Traverser`. From here on out, you'll navigate the API by link relation names. You'll also specify that the API will produce JSON responses with HAL-style hyperlinks so that `Traverser` knows how to parse the incoming resource data. Like `RestTemplate`, you can choose to instantiate a `Traverser` object prior to its use or declare it as a bean to be injected wherever it's needed.

With a `Traverser` object in hand, you can start consuming an API by following links. For example, suppose that you're interested in retrieving a list of all ingredients. You know from section 6.3.1 that the ingredients link has an `href` property that links to the ingredients resource. You'll need to follow that link:

```

ParameterizedTypeReference<Resources<Ingredient>> ingredientType =
    new ParameterizedTypeReference<Resources<Ingredient>>() { };

Resources<Ingredient> ingredientRes =
    traverson
        .follow("ingredients")
        .toObject(ingredientType);

Collection<Ingredient> ingredients = ingredientRes.getContent();

```

By calling the `follow()` method on the `Traverson` object, you can navigate to the resource whose link's relation name is `ingredients`. Now that the client has navigated to `ingredients`, you need to ingest the contents of that resource by calling `toObject()`.

The `toObject()` method requires that you tell it what kind of object to read the data into. This can get a little tricky, considering that you need to read it in as a `Resources<Ingredient>` object, and Java type erasure makes it difficult to provide type information for a generic type. But creating a `ParameterizedTypeReference` helps with that.

As an analogy, imagine that instead of a REST API, this were a homepage on a website. And instead of REST client code, imagine that it's you viewing that homepage in a browser. You see a link on the page that says `Ingredients` and you follow that link by clicking it. Upon arriving at the next page, you read the page, which is analogous to `Traverson` ingesting the content as a `Resources<Ingredient>` object.

Now let's consider a slightly more interesting use case. Let's say that you want to fetch the most recently created tacos. Starting at the home resource, you can navigate to the `recent tacos` resource like this:

```

ParameterizedTypeReference<Resources<Taco>> tacoType =
    new ParameterizedTypeReference<Resources<Taco>>() { };

Resources<Taco> tacoRes =
    traverson
        .follow("tacos")
        .follow("recents")
        .toObject(tacoType);

Collection<Taco> tacos = tacoRes.getContent();

```

Here you follow the `Tacos` link and then, from there, follow the `Recents` link. That brings you to the resource you're interested in, so a call to `toObject()` with an appropriate `ParameterizedTypeReference` gets you what you want. The `.follow()` method can be simplified by listing a trail of relation names to follow:

```

Resources<Taco> tacoRes =
    traverson
        .follow("tacos", "recents")
        .toObject(tacoType);

```

As you can see, Traverson makes easy work of navigating a HATEOAS-enabled API and consuming its resources. But one thing it doesn't do is offer any methods for writing to or deleting from those APIs. In contrast, RestTemplate can write and delete resources, but doesn't make it easy to navigate an API.

When you need to both navigate an API and update or delete resources, you'll need to use RestTemplate and Traverson together. Traverson can still be used to navigate to the link where a new resource will be created. Then RestTemplate can be given that link to do a POST, PUT, DELETE, or any other HTTP request you need.

For example, suppose you want to add a new Ingredient to the Taco Cloud menu. The following `addIngredient()` method teams up Traverson and RestTemplate to post a new Ingredient to the API:

```
private Ingredient addIngredient(Ingredient ingredient) {  
    String ingredientsUrl = traverson  
        .follow("ingredients")  
        .asLink()  
        .getHref();  
  
    return rest.postForObject(ingredientsUrl,  
        ingredient,  
        Ingredient.class);  
}
```

After following the Ingredients link, you ask for the link itself by calling `asLink()`. From that link, you ask for the link's URL by calling `getHref()`. With a URL in hand, you have everything you need to call `postForObject()` on the `RestTemplate` instance and save the new ingredient.

Summary

- Clients can use RestTemplate to make HTTP requests against REST APIs.
- Traverson enables clients to navigate an API using hyperlinks embedded in the responses.



Sending messages asynchronously

This chapter covers

- Asynchronous messaging
- Sending messages with JMS, RabbitMQ, and Kafka
- Pulling messages from a broker
- Listening for messages

It's 4:55 p.m. on Friday. You're minutes away from starting a much-anticipated vacation. You have just enough time to drive to the airport and catch your flight. But before you pack up and head out, you need to be sure your boss and colleagues know the status of the work you've been doing so that on Monday they can pick up where you left off. Unfortunately, some of your colleagues have already skipped out for the weekend, and your boss is tied up in a meeting. What do you do?

The most practical way to communicate your status and still catch your plane is to send a quick email to your boss and your colleagues, detailing your progress and promising to send a postcard. You don't know where they are or when they'll read the email, but you do know they'll eventually return to their desks and read it. Meanwhile, you're on your way to the airport.

Synchronous communication, which is what we've seen with REST, has its place. But it's not the only style of inter-application communication available to developers. *Asynchronous* messaging is a way of indirectly sending messages from one application to another without waiting for a response. This indirection affords looser coupling and greater scalability between the communicating applications.

In this chapter, you're going to use asynchronous messaging to send orders from the Taco Cloud website to a separate application in the Taco Cloud kitchens where the tacos will be prepared. We'll consider three options that Spring offers for asynchronous messaging: the Java Message Service (JMS), RabbitMQ and Advanced Message Queueing Protocol (AMQP), and Apache Kafka. In addition to the basic sending and receiving of messages, we'll look at Spring's support for message-driven POJOs: a way to receive messages that resembles EJB's message-driven beans (MDBs).

8.1 **Sending messages with JMS**

JMS is a Java standard that defines a common API for working with message brokers. First introduced in 2001, JMS has been the go-to approach for asynchronous messaging in Java for a very long time. Before JMS, each message broker had a proprietary API, making an application's messaging code less portable between brokers. But with JMS, all compliant implementations can be worked with via a common interface in much the same way that JDBC has given relational database operations a common interface.

Spring supports JMS through a template-based abstraction known as `JmsTemplate`. Using `JmsTemplate`, it's easy to send messages across queues and topics from the producer side and to receive those messages on the consumer side. Spring also supports the notion of message-driven POJOs: simple Java objects that react to messages arriving on a queue or topic in an asynchronous fashion.

We're going to explore Spring's JMS support, including `JmsTemplate` and message-driven POJOs. But before you can send and receive messages, you need a message broker that's ready to relay those messages between producers and consumers. Let's kick off our exploration of Spring JMS by setting up a message broker in Spring.

8.1.1 **Setting up JMS**

Before you can use JMS, you must add a JMS client to your project's build. With Spring Boot, that couldn't be any easier. All you need to do is add a starter dependency to the build. First, though, you must decide whether you're going to use Apache ActiveMQ, or the newer Apache ActiveMQ Artemis broker.

If you're using ActiveMQ, you'll need to add the following dependency to your project's `pom.xml` file:

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

If ActiveMQ Artemis is the choice, the starter dependency should look like this:

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

Artemis is a next-generation reimplementation of ActiveMQ, effectively making ActiveMQ a legacy option. Therefore, for Taco Cloud you’re going to choose Artemis. But the choice ultimately has little impact on how you’ll write the code that sends and receives messages. The only significant differences will be in how you configure Spring to create connections to the broker.

By default, Spring assumes that your Artemis broker is listening on localhost at port 61616. That’s fine for development purposes, but once you’re ready to send your application into production, you’ll need to set a few properties that tell Spring how to access the broker. The properties you’ll find most useful are listed in table 8.1.

Table 8.1 Properties for configuring the location and credentials of an Artemis broker

Property	Description
spring.artemis.host	The broker’s host
spring.artemis.port	The broker’s port
spring.artemis.user	The user to use to access the broker (optional)
spring.artemis.password	The password to use to access the broker (optional)

For example, consider the following entry from an application.yml file that might be used in a non-development setting:

```
spring:
  artemis:
    host: artemis.tacocloud.com
    port: 61617
    user: tacoweb
    password: 13tm31n
```

This sets up Spring to create broker connections to an Artemis broker listening at artemis.tacocloud.com, port 61617. It also sets the credentials for the application that will be interacting with that broker. The credentials are optional, but they’re recommended for production deployments.

If you were to use ActiveMQ instead of Artemis, you’d need to use the ActiveMQ-specific properties listed in table 8.2.

Table 8.2 Properties for configuring the location and credentials of an ActiveMQ broker

Property	Description
<code>spring.activemq.broker-url</code>	The URL of the broker
<code>spring.activemq.user</code>	The user to use to access the broker (optional)
<code>spring.activemq.password</code>	The password to use to access the broker (optional)
<code>spring.activemq.in-memory</code>	Whether or not to start an in-memory broker (default: <code>true</code>)

Notice that instead of offering separate properties for the broker’s hostname and port, an ActiveMQ broker’s address is specified with a single property, `spring.activemq.broker-url`. The URL should be a `tcp://` URL, as shown in the following YAML snippet:

```
spring:
  activemq:
    broker-url: tcp://activemq.tacocloud.com
    user: tacoweb
    password: l3tm31n
```

Whether you choose Artemis or ActiveMQ, you shouldn’t need to configure these properties for development when the broker is running locally.

If you’re using ActiveMQ, you will, however, need to set the `spring.activemq.in-memory` property to `false` to prevent Spring from starting an in-memory broker. An in-memory broker may seem useful, but it’s only helpful when you’ll be consuming messages from the same application that publishes them (which has limited usefulness).

Instead of using an embedded broker, you’ll want to install and start an Artemis (or ActiveMQ) broker before moving on. Rather than repeat the installation instructions here, I refer you to the broker documentation for details:

- *Artemis*—<https://activemq.apache.org/artemis/docs/latest/using-server.html>
- *ActiveMQ*—<http://activemq.apache.org/getting-started.html#GettingStarted-Pre-InstallationRequirements>

With the JMS starter in your build and a broker waiting to ferry messages from one application to another, you’re ready to start sending messages.

8.1.2 **Sending messages with `JmsTemplate`**

With a JMS starter dependency (either Artemis or ActiveMQ) in your build, Spring Boot will autoconfigure a `JmsTemplate` (among other things) that you can inject and use to send and receive messages.

`JmsTemplate` is the centerpiece of Spring’s JMS integration support. Much like Spring’s other template-oriented components, `JmsTemplate` eliminates a lot of boilerplate code that would otherwise be required to work with JMS. Without `JmsTemplate`, you’d need to write code to create a connection and session with the message broker,

and more code to deal with any exceptions that might be thrown in the course of sending a message. `JmsTemplate` focuses on what you really want to do: send a message.

`JmsTemplate` has several methods that are useful for sending messages, including the following:

```
// Send raw messages
void send(MessageCreator messageCreator) throws JmsException;
void send(Destination destination, MessageCreator messageCreator)
    throws JmsException;
void send(String destinationName, MessageCreator messageCreator)
    throws JmsException;

// Send messages converted from objects
void convertAndSend(Object message) throws JmsException;
void convertAndSend(Destination destination, Object message)
    throws JmsException;
void convertAndSend(String destinationName, Object message)
    throws JmsException;

// Send messages converted from objects with post-processing
void convertAndSend(Object message,
    MessagePostProcessor postProcessor) throws JmsException;
void convertAndSend(Destination destination, Object message,
    MessagePostProcessor postProcessor) throws JmsException;
void convertAndSend(String destinationName, Object message,
    MessagePostProcessor postProcessor) throws JmsException;
```

As you can see, there are really only two methods, `send()` and `convertAndSend()`, each overridden to support different parameters. And if you look closer, you'll notice that the various forms of `convertAndSend()` can be broken into two subcategories. In trying to understand what all of these methods do, consider the following breakdown:

- Three `send()` methods require a `MessageCreator` to manufacture a `Message` object.
- Three `convertAndSend()` methods accept an `Object` and automatically convert that `Object` into a `Message` behind the scenes.
- Three `convertAndSend()` methods automatically convert an `Object` to a `Message`, but also accept a `MessagePostProcessor` to allow for customization of the `Message` before it's sent.

Moreover, each of these three method categories is composed of three overriding methods that are distinguished by how the JMS destination (queue or topic) is specified:

- One method accepts no destination parameter and sends the message to a default destination.
- One method accepts a `Destination` object that specifies the destination for the message.
- One method accepts a `String` that specifies the destination for the message by name.

Putting these methods to work, consider `JmsOrderMessagingService` in the next listing, which uses the most basic form of the `send()` method.

Listing 8.1 Sending an order with `.send()` to a default destination

```
package tacos.messaging;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.JmsTemplate;
import org.springframework.core.MessageCreator;
import org.springframework.stereotype.Service;

@Service
public class JmsOrderMessagingService implements OrderMessagingService {
    private JmsTemplate jms;

    @Autowired
    public JmsOrderMessagingService(JmsTemplate jms) {
        this.jms = jms;
    }

    @Override
    public void sendOrder(Order order) {
        jms.send(new MessageCreator() {
            @Override
            public Message createMessage(Session session)
                throws JMSEException {
                return session.createObjectMessage(order);
            }
        });
    }
}
```

The `sendOrder()` method calls `jms.send()`, passing in an anonymous inner-class implementation of `MessageCreator`. That implementation overrides `createMessage()` to create a new object message from the given `Order` object.

I'm not sure about you, but I think the code in listing 8.1, although straightforward, is a bit clumsy. The ceremony involved in declaring an anonymous inner class complicates an otherwise simple method call. Recognizing that `MessageCreator` is a functional interface, you can tidy up the `sendOrder()` method a bit with a lambda:

```
@Override
public void sendOrder(Order order) {
    jms.send(session -> session.createObjectMessage(order));
}
```

But notice that the call to `jms.send()` doesn't specify a destination. In order for this to work, you must also specify a default destination name with the `spring.jms.template`

.default-destination property. For example, you could set the property in your application.yml file like this:

```
spring:
  jms:
    template:
      default-destination: tacocloud.order.queue
```

In many cases, using a default destination is the easiest choice. It lets you specify the destination name once, allowing the code to only be concerned with sending messages, without regard for where they're being sent. But if you ever need to send a message to a destination other than the default destination, you'll need to specify that destination as a parameter to `send()`.

One way of doing that is by passing a `Destination` object as the first parameter to `send()`. The easiest way to do this is to declare a `Destination` bean and then inject it into the bean that performs messaging. For example, the following bean declares the Taco Cloud order queue `Destination`:

```
@Bean
public Destination orderQueue() {
  return new ActiveMQQueue("tacocloud.order.queue");
}
```

It's important to note that the `ActiveMQQueue` used here is actually from Artemis (from the `org.apache.activemq.artemis.jms.client` package). If you're using ActiveMQ (not Artemis), there's also a class named `ActiveMQQueue` (from the `org.apache.activemq.command` package).

If this `Destination` bean is injected into `JmsOrderMessagingService`, you can use it to specify the destination when calling `send()`:

```
private Destination orderQueue;

@Autowired
public JmsOrderMessagingService(JmsTemplate jms,
                                  Destination orderQueue) {
  this.jms = jms;
  this.orderQueue = orderQueue;
}

...

@Override
public void sendOrder(Order order) {
  jms.send(
    orderQueue,
    session -> session.createObjectMessage(order));
}
```

Specifying the destination with a `Destination` object like this affords you the opportunity to configure the `Destination` with more than just the destination name. But in

practice, you'll almost never specify anything more than the destination name. It's often easier to just send the name as the first parameter to `send()`:

```
@Override  
public void sendOrder(Order order) {  
    jms.send(  
        "tacocloud.order.queue",  
        session -> session.createObjectMessage(order));  
}
```

Although the `send()` method isn't particularly difficult to use (especially when the `MessageCreator` is given as a lambda), a sliver of complexity is added by requiring that you provide a `MessageCreator`. Wouldn't it be simpler if you only needed to specify the object that's to be sent (and optionally the destination)? That describes succinctly how `convertAndSend()` works. Let's take a look.

CONVERTING MESSAGES BEFORE SENDING

`JmsTemplates`'s `convertAndSend()` method simplifies message publication by eliminating the need to provide a `MessageCreator`. Instead, you pass the object that's to be sent directly to `convertAndSend()`, and the object will be converted into a `Message` before being sent.

For example, the following reimplementation of `sendOrder()` uses `convertAndSend()` to send an `Order` to a named destination:

```
@Override  
public void sendOrder(Order order) {  
    jms.convertAndSend("tacocloud.order.queue", order);  
}
```

Just like the `send()` method, `convertAndSend()` will accept either a `Destination` or `String` value to specify the destination, or you can leave out the destination altogether to send the message to the default destination.

Whichever form of `convertAndSend()` you choose, the `Order` passed into `convertAndSend()` is converted into a `Message` before it's sent. Under the covers, this is achieved with an implementation of `MessageConverter` that does the dirty work of converting objects to `Messages`.

CONFIGURING A MESSAGE CONVERTER

`MessageConverter` is a Spring-defined interface that has only two methods to be implemented:

```
public interface MessageConverter {  
    Message toMessage(Object object, Session session)  
        throws JMSEException, MessageConversionException;  
    Object fromMessage(Message message)  
}
```

Although this interface is simple enough to implement, you often won't need to create a custom implementation. Spring already offers a handful of implementations, such as those described in table 8.3.

Table 8.3 Spring message converters for common conversion tasks (all in the org.springframework.jms.support.converter package)

Message converter	What it does
MappingJackson2MessageConverter	Uses the Jackson 2 JSON library to convert messages to and from JSON
MarshallingMessageConverter	Uses JAXB to convert messages to and from XML
MessagingMessageConverter	Converts a Message from the messaging abstraction to and from a Message using an underlying MessageConverter for the payload and a JmsHeaderMapper to map the JMS headers to and from standard message headers
SimpleMessageConverter	Converts Strings to and from TextMessage, byte arrays to and from BytesMessage, Maps to and from MapMessage, and Serializable objects to and from ObjectMessage

SimpleMessageConverter is the default, but it requires that the object being sent implement Serializable. This may be a good idea, but you may prefer to use one of the other message converters, such as MappingJackson2MessageConverter, to avoid that restriction.

To apply a different message converter, all you must do is declare an instance of the chosen converter as a bean. For example, the following bean declaration will enable MappingJackson2MessageConverter to be used instead of SimpleMessageConverter:

```
@Bean
public MappingJackson2MessageConverter messageConverter() {
    MappingJackson2MessageConverter messageConverter =
        new MappingJackson2MessageConverter();
    messageConverter.setTypeIdPropertyName("_ typeId");
    return messageConverter;
}
```

Notice that you called `setTypeIdPropertyName()` on the `MappingJackson2MessageConverter` before returning it. This is very important, as it enables the receiver to know what type to convert an incoming message to. By default, it will contain the fully qualified classname of the type being converted. But that's somewhat inflexible, requiring that the receiver also have the same type, with the same fully qualified classname.

To allow for more flexibility, you can map a synthetic type name to the actual type by calling `setTypeIdMappings()` on the message converter. For example, the

following change to the message converter bean method maps a synthetic order type ID to the Order class:

```

@Bean
public MappingJackson2MessageConverter messageConverter() {
    MappingJackson2MessageConverter messageConverter =
        new MappingJackson2MessageConverter();
    messageConverter.setTypeIdPropertyName("_ typeId");

    Map<String, Class<?>> typeIdMappings = new HashMap<String, Class<?>>();
    typeIdMappings.put("order", Order.class);
    messageConverter.setTypeIdMappings(typeIdMappings);

    return messageConverter;
}

```

Instead of the fully qualified classname being sent in the message's `_ typeId` property, the value `order` will be sent. At the receiving application, a similar message converter will have been configured, mapping `order` to its own understanding of what an order is. That implementation of an order may be in a different package, have a different name, and even have a subset of the sender's `Order` properties.

POST-PROCESSING MESSAGES

Let's suppose that in addition to its lucrative web business, Taco Cloud has decided to open a few brick and mortar taco joints. Given that any of their restaurants could also be a fulfillment center for the web business, they need a way to communicate the source of an order to the kitchens at the restaurants. This will enable the kitchen staff to employ a different process for store orders than for web orders.

It would be reasonable to add a new `source` property to the `Order` object to carry this information, populating it with `WEB` for orders placed online and with `STORE` for orders placed in the stores. But that would require a change to both the website's `Order` class and the kitchen application's `Order` class when, in reality, it's information that's only required for the taco preparers.

An easier solution would be to add a custom header to the message to carry the order's source. If you were using the `send()` method to send the taco orders, this could easily be accomplished by calling `setStringProperty()` on the `Message` object:

```

jms.send("tacocloud.order.queue",
    session -> {
        Message message = session.createObjectMessage(order);
        message.setStringProperty("X_ORDER_SOURCE", "WEB");
    });

```

The problem here is that you aren't using `send()`. By choosing to use `convertAndSend()`, the `Message` object is created under the covers, and you don't have access to it.

Fortunately, there's a way to tweak a `Message` created under the covers before it's sent. By passing in a `MessagePostProcessor` as the final parameter to `convertAndSend()`, you can do whatever you want with the `Message` after it has been created. The

following code still uses `convertAndSend()`, but it also uses a `MessagePostProcessor` to add the `X_ORDER_SOURCE` header before the message is sent:

```
jms.convertAndSend("tacocloud.order.queue", order, new MessagePostProcessor() {
    @Override
    public Message postProcessMessage(Message message) throws JMSException {
        message.setStringProperty("X_ORDER_SOURCE", "WEB");
        return message;
    }
});
```

You may have noticed that `MessagePostProcessor` is a functional interface. This means that you can simplify it a bit by replacing the anonymous inner class with a lambda:

```
jms.convertAndSend("tacocloud.order.queue", order,
    message -> {
        message.setStringProperty("X_ORDER_SOURCE", "WEB");
        return message;
});
```

Although you only need this particular `MessagePostProcessor` for this one call to `convertAndSend()`, you may find yourself using the same `MessagePostProcessor` for several different calls to `convertAndSend()`. In those cases, perhaps a method reference is a better choice than a lambda, avoiding unnecessary code duplication:

```
@GetMapping("/convertAndSend/order")
public String convertAndSendOrder() {
    Order order = buildOrder();
    jms.convertAndSend("tacocloud.order.queue", order,
        this::addOrderSource);
    return "Convert and sent order";
}

private Message addOrderSource(Message message) throws JMSException {
    message.setStringProperty("X_ORDER_SOURCE", "WEB");
    return message;
}
```

You've now seen several ways of sending messages. But it does no good to send a message if nobody ever receives it. Let's look at how you can receive messages with Spring and JMS.

8.1.3 Receiving JMS messages

When it comes to consuming messages, you have the choice of a *pull model*, where your code requests a message and waits until one arrives, or a *push model*, in which messages are handed to your code as they become available.

`JmsTemplate` offers several methods for receiving messages, but all of them use a pull model. You call one of those methods to request a message, and the thread blocks until a message is available (which could be immediately or it might take a while).

On the other hand, you also have the option of using a push model, wherein you define a message listener that's invoked any time a message is available.

Both options are suitable for a variety of use cases. It's generally accepted that the push model is the best choice, as it doesn't block a thread. But in some use cases, a listener could be overburdened if messages arrive too quickly. The pull model enables a consumer to declare that they're ready to process a new message.

Let's look at both ways of receiving messages. We'll start with the pull model offered by `JmsTemplate`.

RECEIVING WITH JMSTEMPLATE

`JmsTemplate` offers several methods for pulling methods from the broker, including the following:

```
Message receive() throws JmsException;  
Message receive(Destination destination) throws JmsException;  
Message receive(String destinationName) throws JmsException;  
  
Object receiveAndConvert() throws JmsException;  
Object receiveAndConvert(Destination destination) throws JmsException;  
Object receiveAndConvert(String destinationName) throws JmsException;
```

As you can see, these six methods mirror the `send()` and `convertAndSend()` methods from `JmsTemplate`. The `receive()` methods receive a raw `Message`, whereas the `receiveAndConvert()` methods use a configured message converter to convert messages into domain types. And for each of these, you can specify either a `Destination` or a `String` containing the destination name, or you can pull a message from the default destination.

To see these in action, you'll write some code that pulls an `Order` from the `taco-cloud.order.queue` destination. The following listing shows `OrderReceiver`, a service component that receives order data using `JmsTemplate.receive()`.

Listing 8.2 Pulling orders from a queue

```
package tacos.kitchen.messaging.jms;  
import javax.jms.Message;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jms.core.JmsTemplate;  
import org.springframework.jms.support.converter.MessageConverter;  
import org.springframework.stereotype.Component;  
  
@Component  
public class JmsOrderReceiver implements OrderReceiver {  
    private JmsTemplate jms;  
    private MessageConverter converter;  
  
    @Autowired  
    public JmsOrderReceiver(JmsTemplate jms, MessageConverter converter) {  
        this.jms = jms;  
        this.converter = converter;  
    }
```

```

public Order receiveOrder() {
    Message message = jms.receive("tacocloud.order.queue");
    return (Order) converter.fromMessage(message);
}
}

```

Here you've used a `String` to specify the destination to pull an order from. The `receive()` method returns an unconverted `Message`. But what you really need is the `Order` that's inside of the `Message`, so the very next thing that happens is that you use an injected message converter to convert the message. The type `ID` property in the message will guide the converter in converting it to an `Order`, but it's returned as an `Object` that requires casting before you can return it.

Receiving a raw `Message` object might be useful in some cases where you need to inspect the message's properties and headers. But often you only need the payload. Converting that payload to a domain type is a two-step process and requires that the message converter be injected into the component. When you only care about the message's payload, `receiveAndConvert()` is a lot simpler. The next listing shows how `JmsOrderReceiver` could be reworked to use `receiveAndConvert()` instead of `receive()`.

Listing 8.3 Receiving an already-converted Order object

```

package tacos.kitchen.messaging.jms;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class JmsOrderReceiver implements OrderReceiver {
    private JmsTemplate jms;

    @Autowired
    public JmsOrderReceiver(JmsTemplate jms) {
        this.jms = jms;
    }

    public Order receiveOrder() {
        return (Order) jms.receiveAndConvert("tacocloud.order.queue");
    }
}

```

This new version of `JmsOrderReceiver` has a `receiveOrder()` method that has been reduced to only one line. And you no longer need to inject a `MessageConverter`, because all of the message conversion will be done behind the scenes in `receiveAndConvert()`.

Before moving on, let's consider how `receiveOrder()` might be used in the Taco Cloud kitchen application. A food preparer at one of Taco Cloud's kitchens might push a button or take some action to indicate that they're ready to start building tacos.

At that point, `receiveOrder()` would be invoked and the call to `receive()` or `receiveAndConvert()` would block. Nothing else would happen until an order message is ready. Once an order arrives, it will be returned from `receiveOrder()` and its information used to display the details of the order for the food preparer to get to work. This seems like a natural choice for a pull model.

Now let's see how a push model works by declaring a JMS listener.

DECLARING MESSAGE LISTENERS

Unlike the pull model, where an explicit call to `receive()` or `receiveAndConvert()` was required to receive a message, a message listener is a passive component that's idle until a message arrives.

To create a message listener that reacts to JMS messages, you simply must annotate a method in a component with `@JmsListener`. The next listing shows a new `OrderListener` component that listens passively for messages, rather than actively requesting them.

Listing 8.4 An OrderListener component that listens for orders

```
package tacos.kitchen.messaging.jms.listener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class OrderListener {
    private KitchenUI ui;

    @Autowired
    public OrderListener(KitchenUI ui) {
        this.ui = ui;
    }

    @JmsListener(destination = "tacocloud.order.queue")
    public void receiveOrder(Order order) {
        ui.displayOrder(order);
    }
}
```

The `receiveOrder()` method is annotated with `JmsListener` to “listen” for messages on the `tacocloud.order.queue` destination. It doesn’t deal with `JmsTemplate`, nor is it explicitly invoked by your application code. Instead, framework code within Spring waits for messages to arrive on the specified destination, and when they arrive, the `receiveOrder()` method is invoked automatically with the message’s `Order` payload as a parameter.

In many ways, the `@JmsListener` annotation is like one of Spring MVC’s request mapping annotations, such as `@GetMapping` or `@PostMapping`. In Spring MVC, methods annotated with one of the request mapping methods react to requests to a specified path. Similarly, methods that are annotated with `@JmsListener` react to messages that arrive in a destination.

Message listeners are often touted as the best choice because they don't block and are able to handle multiple messages quickly. In the context of the Taco Cloud application, however, perhaps they aren't the best choice. The food preparers are a significant bottleneck in the system and may not be able to prepare tacos as quickly as orders come in. A food preparer may have half-fulfilled an order when a new order is displayed on the screen. The kitchen user interface would need to buffer the orders as they arrive to avoid overburdening the kitchen staff.

That's not to say that message listeners are bad. On the contrary, they're a perfect fit when messages can be handled quickly. But when the message handlers need to be able to ask for more messages on their own timing, the pull model offered by JmsTemplate seems more fitting.

Because JMS is defined by a standard Java specification and supported by many message broker implementations, it's a common choice for messaging in Java. But JMS has a few shortcomings, not the least of which is that as a Java specification its use is limited to Java applications. Newer messaging options such as RabbitMQ and Kafka address these shortcomings and are available for other languages and platforms beyond the JVM. Let's set JMS aside and see how you could have implemented your taco order messaging with RabbitMQ.

8.2 Working with RabbitMQ and AMQP

As arguably the most prominent implementation of AMQP, RabbitMQ offers a more advanced message-routing strategy than JMS. Whereas JMS messages are addressed with the name of a destination from which the receiver will retrieve them, AMQP messages are addressed with the name of an exchange and a routing key, which are decoupled from the queue that the receiver is listening to. This relationship between an exchange and queues is illustrated in figure 8.1.

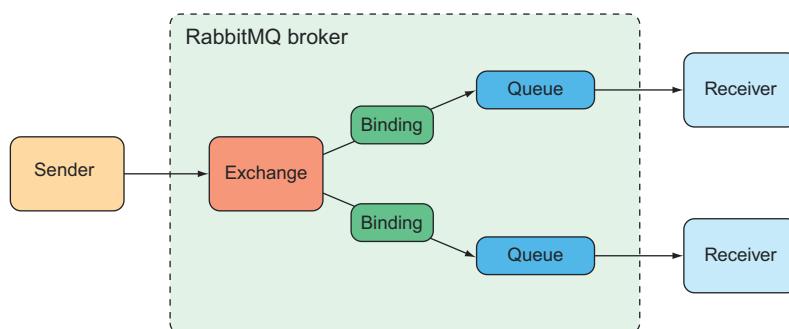


Figure 8.1 Messages sent to a RabbitMQ exchange are routed to one or more queues, based on routing keys and bindings.

When a message arrives at the RabbitMQ broker, it goes to the exchange for which it was addressed. The exchange is responsible for routing it to one or more queues,

depending on the type of exchange, the binding between the exchange and queues, and the value of the message's routing key.

There are several different kinds of exchanges, including the following:

- *Default*—A special exchange that's automatically created by the broker. It routes messages to queues whose name is the same as the message's routing key. All queues will automatically be bound to the default exchange.
- *Direct*—Routes messages to a queue whose binding key is the same as the message's routing key.
- *Topic*—Routes a message to one or more queues where the binding key (which may contain wildcards) matches the message's routing key.
- *Fanout*—Routes messages to all bound queues without regard for binding keys or routing keys.
- *Headers*—Similar to a topic exchange, except that routing is based on message header values rather than routing keys.
- *Dead letter*—A catch-all for any messages that are undeliverable (meaning they don't match any defined exchange-to-queue binding).

The simplest forms of exchanges are default and fanout, as these roughly correspond to a JMS queue and topic. But the other exchanges allow you to define more flexible routing schemes.

The most important thing to understand is that messages are sent to exchanges with routing keys and they're consumed from queues. How they get from an exchange to a queue depends on the binding definitions and what best suits your use cases.

Which exchange type you use and how you define the bindings from exchanges to queues has little bearing on how messages are sent and received in your Spring applications. Therefore we'll focus on how to write code that sends and receives messages with Rabbit.

NOTE For a more detailed discussion on how best to bind queues to exchanges, see *RabbitMQ in Action* by Alvaro Videla and Jason J.W. Williams (Manning, 2012).

8.2.1 Adding RabbitMQ to Spring

Before you can start sending and receiving RabbitMQ messages with Spring, you'll need to add Spring Boot's AMQP starter dependency to your build in place of the Artemis or ActiveMQ starter you added in the previous section:

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

Adding the AMQP starter to your build will trigger autoconfiguration that will create a AMQP connection factory and RabbitTemplate beans, as well as other supporting

components. Simply adding this dependency is all you need to do to start sending and receiving messages from a RabbitMQ broker with Spring. But there are a handful of useful properties you'll want to know about, listed in table 8.4.

Table 8.4 Properties for configuring the location and credentials of a RabbitMQ broker

Property	Description
<code>spring.rabbitmq.addresses</code>	A comma-separated list of RabbitMQ broker addresses
<code>spring.rabbitmq.host</code>	The broker's host (defaults to <code>localhost</code>)
<code>spring.rabbitmq.port</code>	The broker's port (defaults to 5672)
<code>spring.rabbitmq.username</code>	The username to use to access the broker (optional)
<code>spring.rabbitmq.password</code>	The password to use to access the broker (optional)

For development purposes, you'll probably have a RabbitMQ broker that doesn't require authentication running on your local machine, listening on port 5672. These properties likely won't get much use while you're still in development, but they'll no doubt prove useful when your applications move into production.

For example, suppose that as you move into production, your RabbitMQ broker is on a server named `rabbit.tacocloud.com`, listening on port 5673, and requiring credentials. In that case, the following configuration in your `application.yml` file will set those properties when the `prod` profile is active:

```
spring:
  profiles: prod
  rabbitmq:
    host: rabbit.tacocloud.com
    port: 5673
    username: tacoweb
    password: l3tm3ln
```

Now that RabbitMQ is configured in your application, it's time to start sending messages with `RabbitTemplate`.

8.2.2 ***Sending messages with RabbitTemplate***

At the core of Spring's support for RabbitMQ messaging is `RabbitTemplate`. `RabbitTemplate` is similar to `JmsTemplate`, offering a similar set of methods. As you'll see, however, there are some subtle differences that align with the unique way that RabbitMQ works.

With regard to sending messages with `RabbitTemplate`, the `send()` and `convertAndSend()` methods parallel the same-named methods from `JmsTemplate`. But unlike the `JmsTemplate` methods, which only routed messages to a given queue or topic,

RabbitTemplate methods send messages in terms of exchanges and routing keys. Here are a few of the most relevant methods for sending messages with RabbitTemplate:¹

```
// Send raw messages
void send(Message message) throws AmqpException;
void send(String routingKey, Message message) throws AmqpException;
void send(String exchange, String routingKey, Message message)
    throws AmqpException;

// Send messages converted from objects
void convertAndSend(Object message) throws AmqpException;
void convertAndSend(String routingKey, Object message)
    throws AmqpException;
void convertAndSend(String exchange, String routingKey,
    Object message) throws AmqpException;

// Send messages converted from objects with post-processing
void convertAndSend(Object message, MessagePostProcessor mpp)
    throws AmqpException;
void convertAndSend(String routingKey, Object message,
    MessagePostProcessor messagePostProcessor)
    throws AmqpException;
void convertAndSend(String exchange, String routingKey,
    Object message,
    MessagePostProcessor messagePostProcessor)
    throws AmqpException;
```

As you can see, these methods follow a similar pattern to their twins in JmsTemplate. The first three `send()` methods all send a raw `Message` object. The next three `convertAndSend()` methods accept an object that will be converted to a `Message` behind the scenes before being sent. The final three `convertAndSend()` methods are like the previous three, but they accept a `MessagePostProcessor` that can be used to manipulate the `Message` object before it's sent to the broker.

These methods differ from their `JmsTemplate` counterparts in that they accept `String` values to specify an exchange and routing key, rather than a destination name (or `Destination` object). The methods that don't take an exchange will have their messages sent to the default exchange. Likewise, the methods that don't take a routing key will have their messages routed with a default routing key.

Let's put `RabbitTemplate` to work sending taco orders. One way you can do that is by using the `send()` method, as shown in listing 8.5. But before you can call `send()`, you'll need to convert an `Order` object to a `Message`. That could be a tedious job, if not for the fact that `RabbitTemplate` makes its message converter readily available with a `getMessageConverter()` method.

¹ These methods are defined by `AmqpTemplate`, an interface implemented by `RabbitTemplate`.

Listing 8.5 Sending a message with RabbitTemplate.send()

```

package tacos.messaging;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageProperties;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
    org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import tacos.Order;

@Service
public class RabbitOrderMessagingService
    implements OrderMessagingService {
    private RabbitTemplate rabbit;

    @Autowired
    public RabbitOrderMessagingService(RabbitTemplate rabbit) {
        this.rabbit = rabbit;
    }

    public void sendOrder(Order order) {
        MessageConverter converter = rabbit.getMessageConverter();
        MessageProperties props = new MessageProperties();
        Message message = converter.toMessage(order, props);
        rabbit.send("tacocloud.order", message);
    }
}

```

Once you have a `MessageConverter` in hand, it's simple work to convert an `Order` to a `Message`. You must supply any message properties with a `MessageProperties`, but if you don't need to set any such properties, a default instance of `MessageProperties` is fine. Then, all that's left is to call `send()`, passing in the exchange and routing key (both of which are optional) along with the message. In this example, you're specifying only the routing key—`tacocloud.order`—along with the message, so the default exchange will be used.

Speaking of default exchanges, the default exchange name is "" (an empty `String`), which corresponds to the default exchange that's automatically created by the RabbitMQ broker. Likewise, the default routing key is "" (whose routing is dependent upon the exchange and bindings in question). You can override these defaults by setting the `spring.rabbitmq.template.exchange` and `spring.rabbitmq.template.routing-key` properties:

```

spring:
  rabbitmq:
    template:
      exchange: tacocloud.orders
      routing-key: kitchens.central

```

In this case, all messages sent without specifying an exchange will automatically be sent to the exchange whose name is tacocloud.orders. If the routing key is also unspecified in the call to `send()` or `convertAndSend()`, the messages will have a routing key of kitchens.central.

Creating a `Message` object from the message converter is easy enough, but it's even easier to use `convertAndSend()` to let `RabbitTemplate` handle all of the conversion work for you:

```
public void sendOrder(Order order) {  
    rabbit.convertAndSend("tacocloud.order", order);  
}
```

CONFIGURING A MESSAGE CONVERTER

By default, message conversion is performed with `SimpleMessageConverter`, which is able to convert simple types (like `String`) and `Serializable` objects to `Message` objects. But Spring offers several message converters for `RabbitTemplate`, including the following:

- `Jackson2JsonMessageConverter`—Converts objects to and from JSON using the Jackson 2 JSON processor
- `MarshallingMessageConverter`—Converts using a Spring Marshaller and Unmarshaller
- `SerializerMessageConverter`—Converts String and native objects of any kind using Spring's Serializer and Deserializer abstractions
- `SimpleMessageConverter`—Converts String, byte arrays, and `Serializable` types
- `ContentTypeDelegatingMessageConverter`—Delegates to another MessageConverter based on the `contentType` header
- `MessagingMessageConverter`—Delegates to an underlying `MessageConverter` for the message conversion and to an `AmqpHeaderConverter` for the headers

If you need to change the message converter, all you need to do is configure a bean of type `MessageConverter`. For example, for JSON-based message conversion, you can configure a `Jackson2JsonMessageConverter` like this:

```
@Bean  
public MessageConverter messageConverter() {  
    return new Jackson2JsonMessageConverter();  
}
```

Spring Boot autoconfiguration will discover this bean and inject it into `RabbitTemplate` in place of the default message converter.

SETTING MESSAGE PROPERTIES

As with JMS, you may need to set some headers in the messages you send. For example, let's say you need to send an `X_ORDER_SOURCE` for all orders submitted through the Taco Cloud website. When creating your own `Message` objects, you can set the header through the `MessageProperties` instance you give to the message converter.

Revisiting the `sendOrder()` method from listing 8.5, you only need one additional line of code to set the header:

```
public void sendOrder(Order order) {
    MessageConverter converter = rabbit.getMessageConverter();
    MessageProperties props = new MessageProperties();
    props.setHeader("X_ORDER_SOURCE", "WEB");
    Message message = converter.toMessage(order, props);
    rabbit.send("tacocloud.order", message);
}
```

When using `convertAndSend()`, however, you don't have quick access to the `MessageProperties` object. A `MessagePostProcessor` can help you with that, though:

```
@Override
public void sendOrder(Order order) {
    rabbit.convertAndSend("tacocloud.order.queue", order,
        new MessagePostProcessor() {
            @Override
            public Message postProcessMessage(Message message)
                throws AmqpException {
                MessageProperties props = message.getMessageProperties();
                props.setHeader("X_ORDER_SOURCE", "WEB");
                return message;
            }
        });
}
```

Here you supply `convertAndSend()` with an anonymous inner-class implementation of `MessagePostProcessor`. In the `postProcessMessage()` method, you pull the `MessageProperties` from the `Message` and then call `setHeader()` to set the `X_ORDER_SOURCE` header.

Now that you've seen how to send messages with `RabbitTemplate`, let's switch our focus over to the code that receives messages from a `RabbitMQ` queue.

8.2.3 Receiving message from RabbitMQ

You've seen that sending messages with `RabbitTemplate` doesn't differ much from sending messages with `JmsTemplate`. And as it turns out, receiving messages from a `RabbitMQ` queue isn't very different than from `JMS`.

As with JMS, you have two choices:

- Pulling messages from a queue with `RabbitTemplate`
- Having messages pushed to a `@RabbitListener`-annotated method

Let's start by looking at the pull-based `RabbitTemplate.receive()` method.

RECEIVING MESSAGES WITH RABBITTEMPLATE

`RabbitTemplate` comes with several methods for pulling messages from a queue. A few of the most useful ones are listed here:

```
// Receive messages
Message receive() throws AmqpException;
Message receive(String queueName) throws AmqpException;
Message receive(long timeoutMillis) throws AmqpException;
Message receive(String queueName, long timeoutMillis) throws AmqpException;

// Receive objects converted from messages
Object receiveAndConvert() throws AmqpException;
Object receiveAndConvert(String queueName) throws AmqpException;
Object receiveAndConvert(long timeoutMillis) throws AmqpException;
Object receiveAndConvert(String queueName, long timeoutMillis) throws
    AmqpException;

// Receive type-safe objects converted from messages
<T> T receiveAndConvert(ParameterizedTypeReference<T> type) throws
    AmqpException;
<T> T receiveAndConvert(String queueName, ParameterizedTypeReference<T> type)
    throws AmqpException;
<T> T receiveAndConvert(long timeoutMillis, ParameterizedTypeReference<T>
    type) throws AmqpException;
<T> T receiveAndConvert(String queueName, long timeoutMillis,
    ParameterizedTypeReference<T> type)
    throws AmqpException;
```

These methods are the mirror images of the `send()` and `convertAndSend()` methods described earlier. Whereas `send()` is used to send raw `Message` objects, `receive()` receives raw `Message` objects from a queue. Likewise, `receiveAndConvert()` receives messages and uses a message converter to convert them into domain objects before returning them.

But there are a few obvious differences in the method signatures. First, none of these methods take an exchange or routing key as a parameter. That's because exchanges and routing keys are used to route messages to queues, but once the messages are in the queue, their next destination is the consumer who pulls them off the queue. Consuming applications needn't concern themselves with exchanges or routing keys. A queue is the only thing the consuming applications need to know about.

You'll also notice that many of the methods accept a `long` parameter to indicate a timeout for receiving the messages. By default, the receive timeout is 0 milliseconds. That is, a call to `receive()` will return immediately, potentially with a `null` value if no messages are available. This is a marked difference from how the `receive()` methods behave in `JmsTemplate`. By passing in a timeout value, you can have the `receive()` and `receiveAndConvert()` methods block until a message arrives or until the timeout expires. But even with a non-zero timeout, your code will need to be ready to deal with a `null` return.

Let's see how you can put this in action. The next listing shows a new Rabbit-based implementation of `OrderReceiver` that uses `RabbitTemplate` to receive orders.

Listing 8.6 Pulling orders from RabbitMQ with RabbitTemplate

```
package tacos.kitchen.messaging.rabbit;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class RabbitOrderReceiver {
    private RabbitTemplate rabbit;
    private MessageConverter converter;

    @Autowired
    public RabbitOrderReceiver(RabbitTemplate rabbit) {
        this.rabbit = rabbit;
        this.converter = rabbit.getMessageConverter();
    }

    public Order receiveOrder() {
        Message message = rabbit.receive("tacocloud.orders");
        return message != null
            ? (Order) converter.fromMessage(message)
            : null;
    }
}
```

The `receiveOrder()` method is where all of the action takes place. It makes a call to the `receive()` method on the injected `RabbitTemplate` to pull an order from the `tacocloud.orders` queue. It provides no timeout value, so you can only assume that the call returns immediately with either a `Message` or `null`. If a `Message` is returned, you use the `MessageConverter` from the `RabbitTemplate` to convert the `Message` to an `Order`. On the other hand, if `receive()` returns `null`, you'll return a `null`.

Depending on the use case, you may be able to tolerate a small delay. In the Taco Cloud kitchen's overhead display, for example, you can possibly wait a while if no orders are available. Let's say you decide to wait up to 30 seconds before giving up. Then the `receiveOrder()` method can be changed to pass a 30,000 millisecond delay to `receive()`:

```
public Order receiveOrder() {
    Message message = rabbit.receive("tacocloud.order.queue", 30000);
    return message != null
        ? (Order) converter.fromMessage(message)
        : null;
}
```

If you're like me, seeing a hardcoded number like that gives you a bit of discomfort. You might be thinking that it'd be a good idea to create an `@ConfigurationProperties`-annotated class so you could configure that timeout with a Spring Boot configuration

property. I'd agree with you, if it weren't for the fact that Spring Boot already offers such a configuration property. If you want to set the timeout via configuration, simply remove the timeout value in the call to `receive()` and set it in your configuration with the `spring.rabbitmq.template.receive-timeout` property:

```
spring:  
  rabbitmq:  
    template:  
      receive-timeout: 30000
```

Back in the `receiveOrder()` method, notice that you had to use the message converter from `RabbitTemplate` to convert the incoming `Message` object to an `Order` object. But if the `RabbitTemplate` is carrying a message converter around, why can't it do the conversion for you? That's precisely what the `receiveAndConvert()` method is for. Using `receiveAndConvert()`, you can rewrite `receiveOrder()` like this:

```
public Order receiveOrder() {  
  return (Order) rabbit.receiveAndConvert("tacocloud.order.queue");  
}
```

That's a lot simpler, isn't it? The only troubling thing I see is the cast from `Object` to `Order`. There's an alternative to casting, though. Instead, you can pass a `ParameterizedTypeReference` to `receiveAndConvert()` to receive an `Order` object directly:

```
public Order receiveOrder() {  
  return rabbit.receiveAndConvert("tacocloud.order.queue",  
    new ParameterizedTypeReference<Order>() {});  
}
```

It's debatable whether that's better than casting, but it is a more type-safe approach than casting. The only requirement to using a `ParameterizedTypeReference` with `receiveAndConvert()` is that the message converter must be an implementation of `SmartMessageConverter`; `Jackson2JsonMessageConverter` is the only out-of-the-box implementation to choose from.

The pull model offered by `JmsTemplate` fits a lot of use cases, but often it's better to have code that listens for messages and that's invoked when messages arrive. Let's see how you can write message-driven beans that respond to RabbitMQ messages.

HANDLING RABBITMQ MESSAGES WITH LISTENERS

For message-driven RabbitMQ beans, Spring offers `RabbitListener`, the RabbitMQ counterpart to `JmsListener`. To specify that a method should be invoked when a message arrives in a RabbitMQ queue, annotate a bean's method with `@RabbitTemplate`.

For example, the following listing shows a RabbitMQ implementation of `OrderReceiver` that's annotated to listen for order messages rather than to poll for them with `RabbitTemplate`.

Listing 8.7 Declaring a method as a RabbitMQ message listener

```
package tacos.kitchen.messaging.rabbit.listener;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class OrderListener {
    private KitchenUI ui;

    @Autowired
    public OrderListener(KitchenUI ui) {
        this.ui = ui;
    }

    @RabbitListener(queues = "tacocloud.order.queue")
    public void receiveOrder(Order order) {
        ui.displayOrder(order);
    }
}
```

You'll no doubt notice that this looks remarkably like the code from listing 8.4. Indeed, the only thing that changed was the listener annotation—from `@JmsListener` to `@RabbitListener`. As wonderful as `@RabbitListener` is, this near-duplication of code leaves me with little to say about `@RabbitListener` that I haven't already said about `@JmsListener`. They're both great for writing code that responds to messages that are pushed to them from their respective brokers—a JMS broker for `@JmsListener` and a RabbitMQ broker for `@RabbitListener`.

Although you may sense a lack of enthusiasm about `@RabbitListener` in that previous paragraph, be certain that isn't my intent. In truth, the fact that `@RabbitListener` works much like `@JmsListener` is actually quite exciting! It means you don't need to learn a completely different programming model when working with RabbitMQ vs. Artemis or ActiveMQ. The same excitement holds true for the similarities between `RabbitTemplate` and `JmsTemplate`.

Let's hold on to that excitement as we wrap up this chapter by looking at one more messaging option supported by Spring: Apache Kafka.

8.3 **Messaging with Kafka**

Apache Kafka is the newest messaging option we're examining in this chapter. At a glance, Kafka is a message broker just like ActiveMQ, Artemis, or Rabbit. But Kafka has a few unique tricks up its sleeves.

Kafka is designed to run in a cluster, affording great scalability. And by partitioning its topics across all instances in the cluster, it's very resilient. Whereas RabbitMQ deals primarily with queues in exchanges, Kafka utilizes topics only to offer pub/sub messaging.

Kafka topics are replicated across all brokers in the cluster. Each node in the cluster acts as a leader for one or more topics, being responsible for that topic's data and replicating it to the other nodes in the cluster.

Going a step further, each topic can be split into multiple partitions. In that case, each node in the cluster is the leader for one or more partitions of a topic, but not for the entire topic. Responsibility for the topic is split across all nodes. Figure 8.2 illustrates how this works.

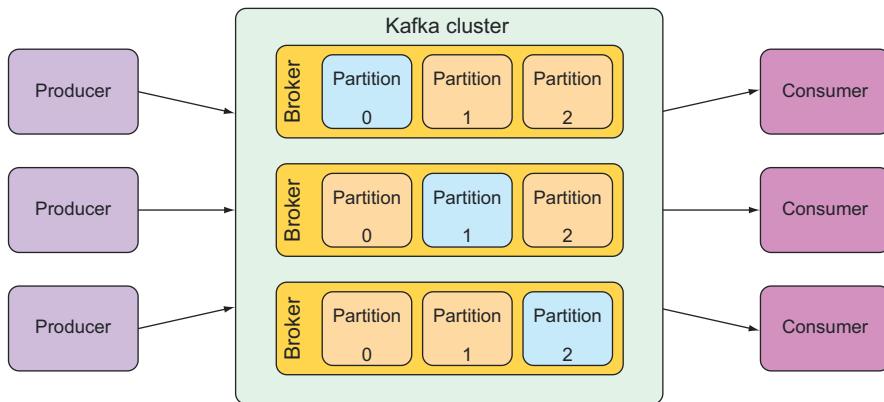


Figure 8.2 A Kafka cluster is composed of multiple brokers, each acting as a leader for partitions of the topics.

Due to Kafka's unique architecture, I encourage you to read more about it in *Kafka in Action* by Dylan Scott (Manning, 2017). For our purposes, we'll focus on how to send messages to and receive them from Kafka with Spring.

8.3.1 Setting up Spring for Kafka messaging

To start using Kafka for messaging, you'll need to add the appropriate dependencies to your build. Unlike the JMS and RabbitMQ options, however, there isn't a Spring Boot starter for Kafka. Have no fear, though; you'll only need one dependency:

```

<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
  
```

This one dependency brings everything you need for Kafka to the project. What's more, its presence will trigger Spring Boot autoconfiguration for Kafka that will, among other things, arrange for a `KafkaTemplate` in the Spring application context. All you need to do is inject the `KafkaTemplate` and go to work sending and receiving messages.

Before you start sending and receiving messages, however, you should be aware of a few properties that will come in handy when working with Kafka. Specifically,

KafkaTemplate defaults to work with a Kafka broker on localhost, listening on port 9092. It's fine to start up a Kafka broker locally while developing an application, but when it's time to go to production, you'll need to configure a different host and port.

The `spring.kafka.bootstrap-servers` property sets the location of one or more Kafka servers used to establish an initial connection to the Kafka cluster. For example, if one of the Kafka servers in the cluster is running at `kafka.tacocloud.com` and listening on port 9092, you can configure its location in YAML like this:

```
spring:
  kafka:
    bootstrap-servers:
      - kafka.tacocloud.com:9092
```

But notice that `spring.kafka.bootstrap-servers` is plural and accepts a list. As such, you can provide it with multiple Kafka servers in the cluster:

```
spring:
  kafka:
    bootstrap-servers:
      - kafka.tacocloud.com:9092
      - kafka.tacocloud.com:9093
      - kafka.tacocloud.com:9094
```

With Kafka set up in your project, you're ready to send and receive messages. You'll start by sending `Order` objects to Kafka using `KafkaTemplate`.

8.3.2 ***Sending messages with KafkaTemplate***

In many ways, `KafkaTemplate` is similar to its JMS and RabbitMQ counterparts. At the same time, it's very different. This becomes apparent as we consider its methods for sending messages:

```
ListenableFuture<SendResult<K, V>> send(String topic, V data);
ListenableFuture<SendResult<K, V>> send(String topic, K key, V data);
ListenableFuture<SendResult<K, V>> send(String topic,
                                         Integer partition, K key, V data);
ListenableFuture<SendResult<K, V>> send(String topic,
                                         Integer partition, Long timestamp, K key, V data);
ListenableFuture<SendResult<K, V>> send(ProducerRecord<K, V> record);
ListenableFuture<SendResult<K, V>> send(Message<?> message);

ListenableFuture<SendResult<K, V>> sendDefault(V data);
ListenableFuture<SendResult<K, V>> sendDefault(K key, V data);
ListenableFuture<SendResult<K, V>> sendDefault(Integer partition,
                                                 K key, V data);
ListenableFuture<SendResult<K, V>> sendDefault(Integer partition,
                                                 Long timestamp, K key, V data);
```

The first thing you may have noticed is that there are no `convertAndSend()` methods. That's because `KafkaTemplate` is typed with generics and is able to deal with domain

types directly when sending messages. In a way, all of the `send()` methods are doing the job of `convertAndSend()`.

You may also have noticed that there are several parameters to `send()` and `sendDefault()` that are quite different from what you used with JMS and Rabbit. When sending messages in Kafka, you can specify the following parameters to guide how the message is sent:

- The topic to send the message to (required for `send()`)
- A partition to write the topic to (optional)
- A key to send on the record (optional)
- A timestamp (optional; defaults to `System.currentTimeMillis()`)
- The payload (required)

The topic and payload are the two most important parameters. Partitions and keys have little effect on how you use `KafkaTemplate`, aside from being extra information provided as parameters to `send()` and `sendDefault()`. For our purposes, we're going to focus on sending the message payload to a given topic and not worry ourselves with partitions and keys.

For the `send()` method, you can also choose to send a `ProducerRecord`, which is little more than a type that captures all of the preceding parameters in a single object. You can also send a `Message` object, but doing so would require you to convert your domain objects into a `Message`. Generally, it's easier to use one of the other methods rather than to create and send a `ProducerRecord` or `Message` object.

Using the `KafkaTemplate` and its `send()` method, you can write a Kafka-based implementation of `OrderMessagingService`. The following listing shows what such an implementation might look like.

Listing 8.8 Sending orders with `KafkaTemplate`

```
package tacos.messaging;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class KafkaOrderMessagingService
    implements OrderMessagingService {

    private KafkaTemplate<String, Order> kafkaTemplate;

    @Autowired
    public KafkaOrderMessagingService(
        KafkaTemplate<String, Order> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    @Override
    public void sendOrder(Order order) {
```

```

    kafkaTemplate.send("tacocloud.orders.topic", order);
}
}

```

In this new implementation of `OrderMessagingService`, the `sendOrder()` method uses the `send()` method of the injected `KafkaTemplate` to send an `Order` to the topic named `tacocloud.orders.topic`. Except for the word “Kafka” scattered through the code, this isn’t much different than the code you wrote for JMS and Rabbit.

If you set a default topic, you can simplify the `sendOrder()` method slightly. First, set your default topic to `tacocloud.orders.topic` by setting the `spring.kafka.template.default-topic` property:

```

spring:
  kafka:
    template:
      default-topic: tacocloud.orders.topic

```

Then, in the `sendOrder()` method, you can call `sendDefault()` instead of `send()` and not specify the topic name:

```

@Override
public void sendOrder(Order order) {
    kafkaTemplate.sendDefault(order);
}

```

Now that your message-sending code has been written, let’s turn our attention to writing code that will receive those messages from Kafka.

8.3.3 Writing Kafka listeners

Aside from the unique method signatures for `send()` and `sendDefault()`, `KafkaTemplate` differs from `JmsTemplate` and `RabbitTemplate` in that it doesn’t offer any methods for receiving messages. That means the only way to consume messages from a Kafka topic using Spring is to write a message listener.

For Kafka, message listeners are defined as methods that are annotated with `@KafkaListener`. The `@KafkaListener` annotation is roughly analogous to `@JmsListener` and `@RabbitListener` and is used in much the same way. The next listing shows what your listener-based order receiver might look like if written for Kafka.

Listing 8.9 Receiving orders with `@KafkaListener`

```

package tacos.kitchen.messaging.kafka.listener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;
import tacos.Order;
import tacos.kitchen.KitchenUI;

```

```

@Component
public class OrderListener {

    private KitchenUI ui;

    @Autowired
    public OrderListener(KitchenUI ui) {
        this.ui = ui;
    }

    @KafkaListener(topics="tacocloud.orders.topic")
    public void handle(Order order) {
        ui.displayOrder(order);
    }
}

```

The `handle()` method is annotated with `@KafkaListener` to indicate that it should be invoked when a message arrives in the topic named `tacocloud.orders.topic`. As it's written in listing 8.9, only an `Order` (the payload) is given to `handle()`. But if you need additional metadata from the message, it can also accept a `ConsumerRecord` or `Message` object.

For example, the following implementation of `handle()` accepts a `ConsumerRecord` so that you can log the partition and timestamp of the message:

```

@KafkaListener(topics="tacocloud.orders.topic")
public void handle(Order order, ConsumerRecord<Order> record) {
    log.info("Received from partition {} with timestamp {}", 
            record.partition(), record.timestamp());
    ui.displayOrder(order);
}

```

Similarly, you could ask for a `Message` instead of a `ConsumerRecord` and achieve the same thing:

```

@KafkaListener(topics="tacocloud.orders.topic")
public void handle(Order order, Message<Order> message) {
    MessageHeaders headers = message.getHeaders();
    log.info("Received from partition {} with timestamp {}", 
            headers.get(KafkaHeaders.RECEIVED_PARTITION_ID)
            .get(KafkaHeaders.RECEIVED_TIMESTAMP));
    ui.displayOrder(order);
}

```

It's worth noting that the message payload is also available via `ConsumerRecord.value()` or `Message.getPayload()`. This means that you could ask for the `Order` through those objects instead of asking for it directly as a parameter to `handle()`.

Summary

- Asynchronous messaging provides a layer of indirection between communicating applications, which allows for looser coupling and greater scalability.
- Spring supports asynchronous messaging with JMS, RabbitMQ, or Apache Kafka.
- Applications can use template-based clients (`JmsTemplate`, `RabbitTemplate`, or `KafkaTemplate`) to send messages via a message broker.
- Receiving applications can consume messages in a pull-based model using the same template-based clients.
- Messages can also be pushed to consumers by applying message listener annotations (`@JmsListener`, `@RabbitListener`, or `@KafkaListener`) to bean methods.

Integrating Spring

This chapter covers

- Processing data in real time
- Defining integration flows
- Using Spring Integration's Java DSL definition
- Integrating with emails, filesystems, and other external systems

One of the most frustrating things I encounter as I travel is being on a long flight and having a poor or nonexistent in-flight internet connection. I like to use my air time to get some work done, including writing many of the pages of this book. If there's no network connection, I'm at a disadvantage if I need to fetch a library or look up a Java Doc, and I'm not able to get much work done. I've learned to pack a book to read for those occasions.

Just as we need to connect to the internet to be productive, many applications must connect to external systems to perform their work. An application may need to read or send emails, interact with an external API, or react to data being written to a database. And, as data is ingested from or written to these external systems, the application may need to process data in some way to translate it to or from the application's own domain.

In this chapter, you'll see how to employ common integration patterns with Spring Integration. Spring Integration is a ready-to-use implementation of many of the integration patterns that are catalogued in *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf (Addison-Wesley, 2003). Each pattern is implemented as a component through which messages ferry data in a pipeline. Using Spring configuration, you can assemble these components into a pipeline through which data flows. Let's get started by defining a simple integration flow that introduces many of the features and characteristics of working with Spring Integration.

9.1 Declaring a simple integration flow

Generally speaking, Spring Integration enables the creation of integration flows through which an application can receive or send data to some resource external to the application itself. One such resource that an application may integrate with is the filesystem. Therefore, among Spring Integration's many components are channel adapters for reading and writing files.

To get your feet wet with Spring Integration, you're going to create an integration flow that writes data to the filesystem. To get started, you need to add Spring Integration to your project build. For Maven, the necessary dependencies are as follows:

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

<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-file</artifactId>
</dependency>
```

The first dependency is the Spring Boot starter for Spring Integration. This dependency is essential to developing a Spring Integration flow, regardless of what the flow may integrate with. Like all Spring Boot starter dependencies, it's available as a check box in the Initializr form.

The second dependency is for Spring Integration's file endpoint module. This module is one of over two dozen endpoint modules used to integrate with external systems. We'll talk more about the endpoint modules in section 9.2.9. But, for now, know that the file endpoint module offers the ability to ingest files from the filesystem into an integration flow and/or to write data from a flow to the filesystem.

Next you need to create a way for the application to send data into an integration flow so that it can be written to a file. To do that, you'll create a gateway interface, such as the one shown next.

Listing 9.1 Message gateway interface to transform method invocations into messages

```
package sia5;
import org.springframework.integration.annotation.MessagingGateway;
```

```
import org.springframework.integration.file.FileHeaders;
import org.springframework.messaging.handler.annotation.Header;

@MessagingGateway(defaultRequestChannel="textInChannel")      ← Declares a
public interface FileWriterGateway {                                message gateway

    void writeToFile(
        @Header(FileHeaders.FILENAME) String filename,   ← Writes to a file
        String data);

}
```

Although it's a simple Java interface, there's a lot to be said about `FileWriterGateway`. The first thing you'll notice is that it's annotated with `@MessagingGateway`. This annotation tells Spring Integration to generate an implementation of this interface at runtime—similar to how Spring Data automatically generates implementations of repository interfaces. Other parts of the code will use this interface when they need to write a file.

The `defaultRequestChannel` attribute of `@MessagingGateway` indicates that any messages resulting from a call to the interface methods should be sent to the given message channel. In this case, you state that any messages that result from a call to `writeToFile()` should be sent to the channel whose name is `textInChannel`.

As for the `writeToFile()` method, it accepts a filename as a `String` and another `String` that is to contain the text that should be written to a file. What's notable about this method signature is that the `filename` parameter is annotated with `@Header`. In this case, the `@Header` annotation indicates that the value passed to `filename` should be placed in a message header (specified as `FileHeaders.FILENAME`, which resolves to `file_name`) rather than in the message payload. The `data` parameter value, on the other hand, is carried in the message payload.

Now that you've a message gateway, you need to configure the integration flow. Although the Spring Integration starter dependency that you added to your build enables essential autoconfiguration for Spring Integration, it's still up to you to write additional configurations to define flows that meet the needs of the application. Three configuration options for declaring integration flows include these:

- XML configuration
- Java configuration
- Java configuration with a DSL

We'll take a look at all three of these configuration styles for Spring Integration, starting with the old-timer, XML configuration.

9.1.1 Defining integration flows with XML

Although I've avoided using XML configuration in this book, Spring Integration has a long history of integration flows defined in XML. Therefore, I think it worthwhile for

me to show at least one example of an XML-defined integration flow. The following listing shows how to configure your sample flow in XML.

Listing 9.2 Defining an integration flow with Spring XML configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:int="http://www.springframework.org/schema/integration"
       xmlns:int-file="http://www.springframework.org/schema/integration/file"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/integration
                           http://www.springframework.org/schema/integration/spring-integration.xsd
                           http://www.springframework.org/schema/integration/file
                           http://www.springframework.org/schema/integration/file/spring-
                           integration-file.xsd">

    <int:channel id="textInChannel" />           ← Declares
                                                textInChannel

    <int:transformer id="upperCase"
                     input-channel="textInChannel"
                     output-channel="fileWriterChannel"
                     expression="payload.toUpperCase()" />   ← Transforms the text

    <int:channel id="fileWriterChannel" />         ← Declares
                                                fileWriterChannel

    <int-file:outbound-channel-adapter id="writer"
                                         channel="fileWriterChannel"
                                         directory="/tmp/sia5/files"
                                         mode="APPEND"
                                         append-new-line="true" />      ← Writes the
                                                text to a file
</beans>
```

Breaking down the XML in listing 9.2:

- You configured a channel named `textInChannel`. You'll recognize this as the same channel that's set as the request channel for `FileWriterGateway`. When the `writeToFile()` method is called on `FileWriterGateway`, the resulting message is published to this channel.
- You configured a transformer that receives messages from `textInChannel`. It uses a Spring Expression Language (SpEL) expression to call `toUpperCase()` on the message payload. The result of the uppercase operation is then published to `fileWriterChannel`.
- You configured the channel named `fileWriterChannel`. This channel serves as the conduit that connects the transformer with the outbound channel adapter.
- Finally, you configured an outbound channel adapter using the `int-file` namespace. This XML namespace is provided by Spring Integration's file module to write files. As you configured it, it receives messages from `fileWriterChannel`

and writes the message payload to a file whose name is specified in the message's `file_name` header in the directory specified in the `directory` attribute. If the file already exists, the file will be appended with a newline rather than overwritten.

This flow is illustrated in figure 9.1 using graphical elements styled after those in *Enterprise Integration Patterns*.

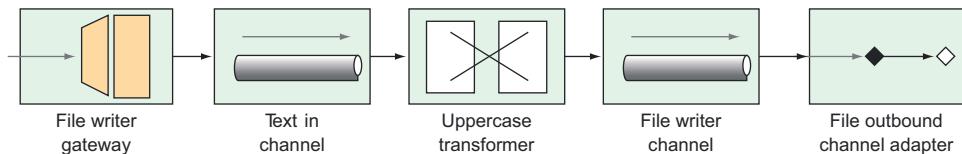


Figure 9.1 The file writer integration flow

If you want to use XML configuration in a Spring Boot application, you'll need to import the XML as a resource into the Spring application. The easiest way to do this is to use Spring's `@ImportResource` annotation on one of your application's Java configuration classes:

```

@Configuration
@ImportResource("classpath:/filewriter-config.xml")
public class FileWriterIntegrationConfig { ... }
  
```

Although XML-based configuration has served Spring Integration well, most developers have grown wary of using XML. (And, as I said, I'm avoiding XML configuration in this book.) Let's set aside those angle brackets and turn our attention to Spring Integration's Java configuration style.

9.1.2 Configuring integration flows in Java

Most modern Spring applications have eschewed XML configuration in favor of Java configuration. In fact, in Spring Boot applications, Java configuration is a natural style to complement autoconfiguration. Therefore, if you're adding an integration flow to a Spring Boot application, it makes perfect sense to define the flow in Java.

As a sample of how to write an integration flow with Java configuration, take a look at the next listing. This shows the same file-writing integration flow as before, but this time it's written in Java.

Listing 9.3 Using Java configuration to define an integration flow

```

package sia5;
import java.io.File;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.annotation.Transformer;
import org.springframework.integration.file.FileWriterMessageHandler;
  
```

```

import org.springframework.integration.file.support.FileExistsMode;
import org.springframework.integration.transformer.GenericTransformer;

@Configuration
public class FileWriterIntegrationConfig {

    @Bean
    @Transformer(inputChannel="textInChannel",           ← Declares a
                  outputChannel="fileWriterChannel")   transformer
    public GenericTransformer<String, String> upperCaseTransformer() {
        return text -> text.toUpperCase();
    }

    @Bean
    @ServiceActivator(inputChannel="fileWriterChannel") ← Declares a
    public FileWritingMessageHandler fileWriter() {     file writer
        FileWritingMessageHandler handler =
            new FileWritingMessageHandler(new File("/tmp/sia5/files"));
        handler.setExpectReply(false);
        handler.setFileExistsMode(FileExistsMode.APPEND);
        handler.setAppendNewLine(true);
        return handler;
    }

}

```

With Java configuration, you declare two beans: a transformer and a file-writing message handler. The transformer is a `GenericTransformer`. Because `GenericTransformer` is a functional interface, you're able to provide its implementation as a lambda that calls `toUpperCase()` on the message text. The transformer bean is annotated with `@Transformer` designating it as a transformer in the integration flow that receives messages on a channel named `textInChannel` and writes messages to the channel named `fileWriterChannel`.

As for the file-writing bean, it's annotated with `@ServiceActivator` to indicate that it'll accept messages from `fileWriterChannel` and hand those messages over to the service defined by an instance of `FileWritingMessageHandler`. `FileWritingMessageHandler` is a message handler that writes a message payload to a file in a specified directory using a filename specified in the message's `file_name` header. As with the XML example, `FileWritingMessageHandler` is configured to append to the file with a newline.

One thing unique about the configuration of the `FileWritingMessageHandler` bean is that there's a call to `setExpectReply(false)` to indicate that the service activator shouldn't expect a reply channel (a channel through which a value may be returned to upstream components in the flow). If you don't call `setExpectReply()`, the file-writing bean defaults to `true` and, although the pipeline still functions as expected, you'll see a few errors logged stating that no reply channel was configured.

You'll also notice that you didn't need to explicitly declare the channels. The `textInChannel` and `fileWriterChannel` channels will be created automatically if no beans

with those names exist. But if you want more control over how the channels are configured, you can explicitly construct them as beans like this:

```
@Bean
public MessageChannel textInChannel() {
    return new DirectChannel();
}

...

@Bean
public MessageChannel fileWriterChannel() {
    return new DirectChannel();
}
```

The Java configuration option is arguably easier to read and slightly briefer, and is certainly consistent with the Java-only configuration I'm shooting for in this book. But it can be made even more streamlined with Spring Integration's Java DSL (domain-specific language) configuration style.

9.1.3 Using Spring Integration's DSL configuration

Let's take one more stab at defining the file-writing integration flow. This time, you'll still define it in Java, but you'll use Spring Integration's Java DSL. Rather than declare an individual bean for each component in the flow, you'll declare a single bean that defines the entire flow.

Listing 9.4 Providing a fluent API for designing integration flows

```
package sia5;
import java.io.File;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.channel.MessageChannels;
import org.springframework.integration.file.dsl.Files;
import org.springframework.integration.file.support.FileExistsMode;

@Configuration
public class FileWriterIntegrationConfig {

    @Bean
    public IntegrationFlow fileWriterFlow() {
        return IntegrationFlows
            .from(MessageChannels.direct("textInChannel"))
            .<String, String>transform(t -> t.toUpperCase())
            .handle(Files
                .outboundAdapter(new File("/tmp/sia5/files"))
                .fileExistsMode(FileExistsMode.APPEND)
                .appendNewLine(true))
            .get();
    }
}
```

The diagram shows three annotations from Listing 9.4 with corresponding callouts:

- `@Configuration`: A callout labeled "Inbound channel" points to the annotation.
- `.from(MessageChannels.direct("textInChannel"))`: A callout labeled "Declares a transformer" points to this line.
- `.handle(Files` block: A callout labeled "Handles writing to a file" points to the start of this block.

This new configuration is as terse as it can possibly be, capturing the entire flow in a single bean method. The `IntegrationFlows` class initiates the builder API, from which you can declare the flow.

In listing 9.4, you start by receiving messages from the channel named `textInChannel`, which then go to a transformer that uppercases the message payload. After the transformer, messages are handled by an outbound channel adapter created from the `Files` type provided in Spring Integration's file module. Finally, a call to `get()` builds the `IntegrationFlow` to be returned. In short, this one bean method defines the same integration flow as the XML and Java configuration examples.

You'll notice that, as with the Java configuration example, you don't need to explicitly declare channel beans. Although you reference `textInChannel`, it's automatically created by Spring Integration because there's no existing channel bean with that name. But you can explicitly declare the channel bean if you want.

As for the channel that connects the transformer to the outbound channel adapter, you don't even reference it by name. If there's a need to explicitly configure the channel, you can reference it by name in the flow definition with a call to `channel()`:

```
@Bean
public IntegrationFlow fileWriterFlow() {
    return IntegrationFlows
        .from(MessageChannels.direct("textInChannel"))
        .<String, String>transform(t -> t.toUpperCase())
        .channel(MessageChannels.direct("fileWriterChannel"))
        .handle(Files
            .outboundAdapter(new File("/tmp/sia5/files"))
            .fileExistsMode(FileExistsMode.APPEND)
            .appendNewLine(true))
        .get();
}
```

One thing to keep in mind when working with Spring Integration's Java DSL (as with any fluent API) is that you must employ whitespace shrewdly to maintain readability. In the example given here, I've been careful to indent lines to indicate blocks of related code. For even longer, more complex flows, you may even consider extracting portions of the flow into separate methods or subflows for better readability.

Now that you've seen a simple flow defined using three different configuration styles, let's step back and take a look at Spring Integration's big picture.

9.2 **Surveying the Spring Integration landscape**

Spring Integration covers a lot of ground with a multitude of integration scenarios. Trying to include all of it in a single chapter would be like trying to fit an elephant in an envelope. Instead of a comprehensive treatment of Spring Integration, I'll present a photograph of the Spring Integration elephant to give you some idea of how it works. Then you'll create one more integration flow that adds functionality to the Taco Cloud application.

An integration flow is composed of one or more of the following components. Before you write any more code, we'll take a brief look at the role each of these components plays in an integration flow:

- *Channels*—Pass messages from one element to another.
- *Filters*—Conditionally allow messages to pass through the flow based on some criteria.
- *Transformers*—Change message values and/or convert message payloads from one type to another.
- *Routers*—Direct messages to one of several channels, typically based on message headers.
- *Splitters*—Split incoming messages into two or more messages, each sent to different channels.
- *Aggregators*—The opposite of splitters, combining multiple messages coming in from separate channels into a single message.
- *Service activators*—Hand a message off to some Java method for processing, and then publish the return value on an output channel.
- *Channel adapters*—Connect a channel to some external system or transport. Can either accept input or write to the external system.
- *Gateways*—Pass data into an integration flow via an interface.

You've already seen a few of these components in play when you defined the file-writing integration flow. The `FileWriterGateway` interface was the gateway through which an application submitted text to be written to a file. You also defined a transformer to convert the given text to uppercase; then you declared a service gateway that performed the task of writing the text to a file. And the flow had two channels, `textInChannel` and `fileWriterChannel`, that connected the other components with each other. Now, a quick tour of the integration flow components, as promised.

9.2.1 Message channels

Message channels are the means by which messages move through an integration pipeline (figure 9.2). They're the pipes that connect all the other parts of Spring Integration plumbing together.

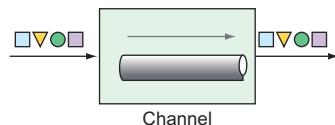


Figure 9.2 Message channels are conduits through which data flows between other components in an integration flow.

Spring Integration provides several channel implementations, including these:

- `PublishSubscribeChannel`—Messages published into a `PublishSubscribeChannel` are passed on to one or more consumers. If there are multiple consumers, all of them receive the message.

- **QueueChannel**—Messages published into a QueueChannel are stored in a queue until pulled by a consumer in a first in, first out (FIFO) fashion. If there are multiple consumers, only one of them receives the message.
- **PriorityChannel**—Like QueueChannel but, rather than FIFO behavior, messages are pulled by consumers based on the message priority header.
- **RendezvousChannel**—Like QueueChannel except that the sender blocks the channel until a consumer receives the message, effectively synchronizing the sender with the consumer.
- **DirectChannel**—Like PublishSubscribeChannel but sends a message to a single consumer by invoking the consumer in the same thread as the sender. This allows for transactions to span across the channel.
- **ExecutorChannel**—Similar to DirectChannel but the message dispatch occurs via a TaskExecutor, taking place in a separate thread from the sender. This channel type doesn't support transactions that span the channel.
- **FluxMessageChannel**—A Reactive Streams Publisher message channel based on Project Reactor's Flux. (We'll talk more about Reactive Streams, Reactor, and Flux in chapter 10.)

In both the Java configuration and Java DSL styles, input channels are automatically created, with DirectChannel as the default. But if you want to use a different channel implementation, you'll need to explicitly declare the channel as a bean and reference it in the integration flow. For example, to declare a PublishSubscribeChannel, you'd declare the following @Bean method:

```
@Bean
public MessageChannel orderChannel() {
    return new PublishSubscribeChannel();
}
```

Then you'd reference this channel by name in the integration flow definition. For example, if the channel were being consumed by a service activator bean, you'd reference it in the `inputChannel` attribute of `@ServiceActivator`:

```
@ServiceActivator(inputChannel="orderChannel")
```

Or, if you're using the Java DSL configuration style, you'd reference it with a call to `channel()`:

```
@Bean
public IntegrationFlow orderFlow() {
    return IntegrationFlows
        ...
        .channel("orderChannel")
        ...
        .get();
}
```

It's important to note that if you're using `QueueChannel`, the consumers must be configured with a poller. For instance, suppose that you've declared a `QueueChannel` bean like this:

```
@Bean
public MessageChannel orderChannel() {
    return new QueueChannel();
}
```

You'd need to make sure that the consumer is configured to poll the channel for messages. In the case of a service activator, the `@ServiceActivator` annotation might look like this:

```
@ServiceActivator(inputChannel="orderChannel",
    poller=@Poller(fixedRate="1000"))
```

In this example, the service activator polls from the channel named `orderChannel` every 1 second (or 1,000 ms).

9.2.2 Filters

Filters can be placed in the midst of an integration pipeline to allow or disallow messages from proceeding to the next step in the flow (figure 9.3).

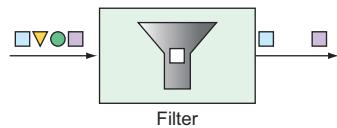


Figure 9.3 Filters based on some criteria allow or disallow messages from proceeding in the pipeline.

For example, suppose that messages containing integer values are published through a channel named `numberChannel`, but you only want even numbers to pass on to the channel named `evenNumberChannel`. In that case, you could declare a filter with the `@Filter` annotation like this:

```
@Filter(inputChannel="numberChannel",
    outputChannel="evenNumberChannel")
public boolean evenNumberFilter(Integer number) {
    return number % 2 == 0;
}
```

Alternatively, if you're using the Java DSL configuration style to define your integration flow, you could make a call to `filter()` like this:

```
@Bean
public IntegrationFlow evenNumberFlow(AtomicInteger integerSource) {
    return IntegrationFlows
        ...
        .<Integer>filter((p) -> p % 2 == 0)
        ...
}
```

```

        .get();
}
}

```

In this case, you use a lambda to implement the filter. But, in truth, the `filter()` method accepts a `GenericSelector` as an argument. This means that you can implement the `GenericSelector` interface instead, should your filtering needs be too involved for a simple lambda.

9.2.3 Transformers

Transformers perform some operation on messages, typically resulting in a different message and, possibly, with a different payload type (see figure 9.4). The transformation can be something simple, such as performing mathematic operations on a number or manipulating a `String` value. Or the transformation can be more complex, such as using a `String` value representing an ISBN to look up and return details of the corresponding book.

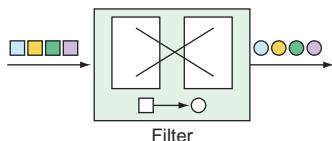


Figure 9.4 Transformers morph messages as they flow through an integration flow.

For example, suppose that integer values are being published on a channel named `numberChannel` and you want to convert those numbers to a `String` containing the Roman numeral equivalent. In that case, you can declare a bean of type `GenericTransformer` and annotate it with `@Transformer` as follows:

```

@Bean
@Transformer(inputChannel="numberChannel",
             outputChannel="romanNumberChannel")
public GenericTransformer<Integer, String> romanNumTransformer() {
    return RomanNumbers::toRoman;
}

```

The `@Transformer` annotation designates this bean as a transformer bean that receives `Integer` values from the channel named `numberChannel` and uses a static method named `toRoman()` to do the conversion. (The `toRoman()` method is statically defined in a class named `RomanNumbers` and referenced here with a method reference.) The result is published to the channel named `romanNumberChannel`.

In the Java DSL configuration style, it's even easier with a call to `transform()`, passing in the method reference to the `toRoman()` method:

```

@Bean
public IntegrationFlow transformerFlow() {
    return IntegrationFlows
        ...
        .transform(RomanNumbers::toRoman)
}

```

```

    ...
    .get();
}
}

```

Although you've used a method reference in both of the transformer code samples, know that the transformer can also be specified as a lambda. Or, if the transformer is complex enough to warrant a separate Java class, you can inject it as a bean into the flow configuration and pass the reference to the `transform()` method:

```

@Bean
public RomanNumberTransformer romanNumberTransformer() {
    return new RomanNumberTransformer();
}

@Bean
public IntegrationFlow transformerFlow(
    RomanNumberTransformer romanNumberTransformer) {
    return IntegrationFlows
        ...
        .transform(romanNumberTransformer)
        ...
        .get();
}

```

Here, you declare a bean of type `RomanNumberTransformer`, which itself is an implementation of Spring Integration's `Transformer` or `GenericTransformer` interfaces. The bean is injected into the `transformerFlow()` method and passed to the `transform()` method when defining the integration flow.

9.2.4 Routers

Routers, based on some routing criteria, allow for branching in an integration flow, directing messages to different channels (see figure 9.5).

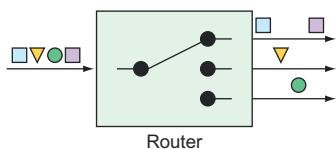


Figure 9.5 Routers direct messages to different channels, based on some criteria applied to the messages.

For example, suppose that you have a channel named `numberChannel` through which integer values flow. And let's say that you want to direct all messages with even numbers to a channel named `evenChannel`, while messages with odd numbers are routed to a channel named `oddChannel`. To create such a routing in your integration flow, you can declare a bean of type `AbstractMessageRouter` and annotate the bean with `@Router`:

```

@Bean
@Router(inputChannel="numberChannel")
public AbstractMessageRouter evenOddRouter() {
}

```

```

        return new AbstractMessageRouter() {
            @Override
            protected Collection<MessageChannel>
                determineTargetChannels(Message<?> message) {
                Integer number = (Integer) message.getPayload();
                if (number % 2 == 0) {
                    return Collections.singleton(evenChannel());
                }
                return Collections.singleton(oddChannel());
            }
        };
    }

    @Bean
    public MessageChannel evenChannel() {
        return new DirectChannel();
    }

    @Bean
    public MessageChannel oddChannel() {
        return new DirectChannel();
    }
}

```

The `AbstractMessageRouter` bean declared here accepts messages from an input channel named `numberChannel`. The implementation, defined as an anonymous inner class, examines the message payload and, if it's an even number, returns the channel named `evenChannel` (declared as a bean after the router bean). Otherwise, the number in the channel payload must be odd; in which case, the channel named `oddChannel` is returned (also declared in a bean declaration method).

In Java DSL form, routers are declared by calling `route()` in the course of a flow definition, as shown here:

```

@Bean
public IntegrationFlow numberRoutingFlow(AtomicInteger source) {
    return IntegrationFlows
        ...
        .<Integer, String>route(n -> n%2==0 ? "EVEN":"ODD", mapping -> mapping
            .subFlowMapping("EVEN", sf -> sf
                .<Integer, Integer>transform(n -> n * 10)
                .handle((i,h) -> { ... })
            )
            .subFlowMapping("ODD", sf -> sf
                .transform(RomanNumbers::toRoman)
                .handle((i,h) -> { ... })
            )
        )
        .get();
}

```

Although it's still possible to declare an `AbstractMessageRouter` and pass it into `route()`, this example uses a lambda to determine if a message payload is odd or even.

If it's even, then a `String` value of `EVEN` is returned. If it's odd, then `ODD` is returned. These values are then used to determine which submapping will handle the message.

9.2.5 Splitters

At times, in an integration flow it can be useful to split a message into multiple messages to be handled independently. Splitters, as illustrated in figure 9.6, will split and handle those messages for you.

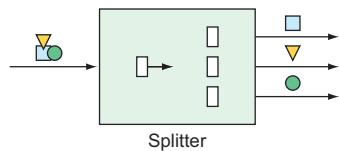


Figure 9.6 Splitters break down messages into two or more separate messages that can be handled by separate subflows.

Splitters are useful in many circumstances, but there are two essential use cases for which you might use a splitter:

- *A message payload contains a collection of items of the same type that you'd like to process as individual message payloads.* For example, a message carrying a list of products might be split into multiple messages with payloads of one product each.
- *A message payload carries information that, although related, can be split into two or more messages of different types.* For example, a purchase order might carry delivery, billing, and line-item information. The delivery details might be processed by one subflow, billing by another, and line items in yet another. In this use case, the splitter is typically followed by a router that routes messages by payload type to ensure that the data gets handled by the right subflow.

When splitting a message payload into two or more messages of different types, it's usually sufficient to define a POJO that extracts the individual pieces of the incoming payload and returns them as elements of a collection.

For example, suppose that you want to split a message carrying a purchase order into two messages: one carrying the billing information and another carrying a list of line items. The following `OrderSplitter` will do the job:

```

public class OrderSplitter {
    public Collection<Object> splitOrderIntoParts(PurchaseOrder po) {
        ArrayList<Object> parts = new ArrayList<>();
        parts.add(po.getBillInfo());
        parts.add(po.getLineItems());
        return parts;
    }
}
  
```

You can then declare an `OrderSplitter` bean as part of the integration flow by annotating it with `@Splitter` like this:

```

@Bean
@Splitter(inputChannel="poChannel",
          outputChannel="splitOrderChannel")
public OrderSplitter orderSplitter() {
    return new OrderSplitter();
}

```

Here, purchase orders arrive on the channel named `poChannel` and are split by `OrderSplitter`. Then, each item in the returned collection is published as a separate message in the integration flow to a channel named `splitOrderChannel`. At this point in the flow, you can declare a `PayloadTypeRouter` to route the billing information and the line items to their own subflow:

```

@Bean
@Router(inputChannel="splitOrderChannel")
public MessageRouter splitOrderRouter() {
    PayloadTypeRouter router = new PayloadTypeRouter();
    router.setChannelMapping(
        BillingInfo.class.getName(), "billingInfoChannel");
    router.setChannelMapping(
        List.class.getName(), "lineItemsChannel");
    return router;
}

```

As its name implies, `PayloadTypeRouter` routes messages to different channels based on their payload type. As configured here, messages whose payload is of type `BillingInfo` are routed to a channel named `billingInfoChannel` for further processing. As for the line items, they're in a `java.util.List` collection; therefore, you mapped payloads of type `List` to be routed to a channel named `lineItemsChannel`.

As things stand, the flow splits into two subflows: one through which `BillingInfo` objects flow and another through which a `List<LineItem>` flows. But what if you want to break it down further such that instead of dealing with a `List` of `LineItems`, you process each `LineItem` separately? All you need to do to split the line-item list into multiple messages, one for each line item, is write a method (not a bean) that's annotated with `@Splitter` and returns a collection of `LineItems`, perhaps something like this:

```

@Splitter(inputChannel="lineItemsChannel", outputChannel="lineItemChannel")
public List<LineItem> lineItemSplitter(List<LineItem> lineItems) {
    return lineItems;
}

```

When a message carrying a payload of `List<LineItem>` arrives in the channel named `lineItemsChannel`, it passes into the `lineItemSplitter()` method. Per the rules of a splitter, the method must return a collection of the items to be split. In this case, you already have a collection of `LineItems`, so you just return the collection directly. As a result, each `LineItem` in the collection is published in a message of its own to the channel named `lineItemChannel`.

If you'd rather use the Java DSL to declare the same splitter/router configuration, you can do so with calls to `split()` and `route()`:

```
return IntegrationFlows
    ...
    .split(orderSplitter())
    . route(
        p -> {
            if (p.getClass().isAssignableFrom(BillingInfo.class)) {
                return "BILLING_INFO";
            } else {
                return "LINE_ITEMS";
            }
        }, mapping -> mapping
        .subFlowMapping("BILLING_INFO", sf -> sf
            .<BillingInfo> handle((billingInfo, h) -> {
                ...
            }))
        .subFlowMapping("LINE_ITEMS", sf -> sf
            .split()
            .<LineItem> handle((lineItem, h) -> {
                ...
            }))
    )
    .get();
}
```

The DSL form of the flow definition is certainly terser, if not arguably a bit more difficult to follow. It uses the same `OrderSplitter` to split the order as the Java configuration example. After the order is split, it's routed by its type to two separate subflows.

9.2.6 Service activators

Service activators receive messages from an input channel and send those messages to an implementation of `MessageHandler`, as shown in figure 9.7.

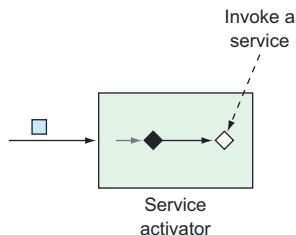


Figure 9.7 Service activators invoke some service by way of a `MessageHandler` on receipt of a message.

Spring Integration offers several `MessageHandler` implementations out of the box (even `PayloadTypeRouter` is an implementation of `MessageHandler`), but you'll often need to provide some custom implementation to act as a service activator. As an example, the following code shows how to declare a `MessageHandler` bean, configured to be a service activator:

```

@Bean
@ServiceActivator(inputChannel="someChannel")
public MessageHandler sysoutHandler() {
    return message -> {
        System.out.println("Message payload: " + message.getPayload());
    };
}

```

The bean is annotated with `@ServiceActivator` to designate it as a service activator that handles messages from the channel named `someChannel`. As for the `MessageHandler` itself, it's implemented via a lambda. Although it's a simple `MessageHandler`, when given a `Message`, it emits its payload to the standard output stream.

Alternatively, you could declare a service activator that processes the data in the incoming message before returning a new payload. In that case, the bean should be a `GenericHandler` rather than a `MessageHandler`:

```

@Bean
@ServiceActivator(inputChannel="orderChannel",
                   outputChannel="completeOrder")
public GenericHandler<Order> orderHandler(
    OrderRepository orderRepo) {
    return (payload, headers) -> {
        return orderRepo.save(payload);
    };
}

```

In this case, the service activator is a `GenericHandler` that expects messages with a payload of type `Order`. When the order arrives, it's saved via a repository; the resulting saved `Order` is returned to be sent to the output channel whose name is `completeChannel`.

You may notice that a `GenericHandler` is given not only the payload, but also the message headers (even if the example doesn't use those headers in any way). If you prefer, you can also use service activators in the Java DSL configuration style by passing a `MessageHandler` or `GenericHandler` to `handle()` in the flow definition:

```

public IntegrationFlow someFlow() {
    return IntegrationFlows
        ...
        .handle(msg -> {
            System.out.println("Message payload: " + msg.getPayload());
        })
        .get();
}

```

In this case, the `MessageHandler` is given as a lambda, but you could also provide it as a method reference or even as an instance of a class that implements the `MessageHandler` interface. If you give it a lambda or method reference, be aware that it accepts a message as a parameter.

Similarly, `handle()` can be written to accept a `GenericHandler` if the service activator isn't intended to be the end of the flow. Applying the order-saving service activator from before, you could configure the flow with the Java DSL like this:

```
public IntegrationFlow orderFlow(OrderRepository orderRepo) {
    return IntegrationFlows
        ...
        .<Order>handle((payload, headers) -> {
            return orderRepo.save(payload);
        })
        ...
        .get();
}
```

When working with a `GenericHandler`, the lambda or method reference accepts the message payload and headers as parameters. Also, if you choose to use `GenericHandler` at the end of a flow, you'll need to return `null`, or else you'll get errors indicating that there's no output channel specified.

9.2.7 Gateways

Gateways are the means by which an application can submit data into an integration flow and, optionally, receive a response that's the result of the flow. Implemented by Spring Integration, gateways are realized as interfaces that the application can call to send messages to the integration flow (figure 9.8).

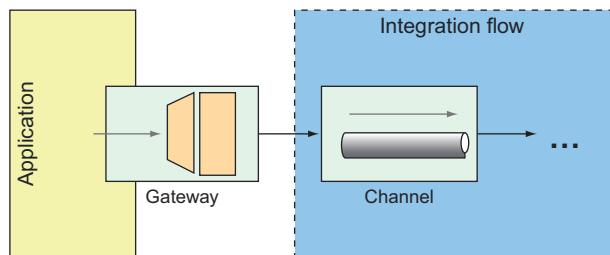


Figure 9.8 Service gateways are interfaces through which an application can submit messages to an integration flow.

You've already seen an example of a message gateway with `FileWriterGateway`. `FileWriterGateway` was a one-way gateway with a method accepting a `String` to write to a file, returning `void`. It's just about as easy to write a two-way gateway. When writing the gateway interface, be sure that the method returns some value to publish into the integration flow.

As an example, imagine a gateway that fronts a simple integration flow that accepts a `String` and translates the given `String` to all uppercase. The gateway interface might look something like this:

```
package com.example.demo;
import org.springframework.integration.annotation.MessagingGateway;
import org.springframework.stereotype.Component;
```

```

@Component
@MessagingGateway(defaultRequestChannel="inChannel",
                  defaultReplyChannel="outChannel")
public interface UpperCaseGateway {
    String uppercase(String in);
}

```

What's amazing about this interface is that it's not necessary to implement it. Spring Integration automatically provides an implementation at runtime that sends and receives data through the specified channels.

When `uppercase()` is called, the given `String` is published to the integration flow into the channel named `inChannel`. And, regardless of how the flow is defined or what it does, when data arrives in the channel named `outChannel`, it's returned from the `uppercase()` method.

As for the uppercase integration flow, it's a simplistic integration flow with only a single step to transform the `String` to uppercase. Here, it's expressed in the Java DSL configuration:

```

@Bean
public IntegrationFlow uppercaseFlow() {
    return IntegrationFlows
        .from("inChannel")
        .<String, String> transform(s -> s.toUpperCase())
        .channel("outChannel")
        .get();
}

```

As defined here, the flow starts with data coming into the channel named `inChannel`. The message payload is then transformed by the transformer, which is defined here as a lambda expression, to perform an uppercase operation. The resulting message is then published to the channel named `outChannel`, which is what you've declared as the reply channel for the `UpperCaseGateway` interface.

9.2.8 Channel adapters

Channel adapters represent the entry and exit points of an integration flow. Data enters an integration flow by way of an inbound channel adapter and exits an integration flow by way of an outbound channel adapter. This is illustrated in figure 9.9.

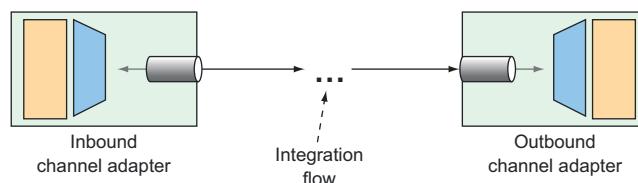


Figure 9.9 Channel adapters are the entry and exit points of an integration flow.

Inbound channel adapters can take many forms, depending on the source of the data they introduce into the flow. For example, you might declare an inbound channel

adapter that introduces incrementing numbers from an AtomicInteger into the flow. Using Java configuration, it might look like this:

```
@Bean
@InboundChannelAdapter(
    poller=@Poller(fixedRate="1000"), channel="numberChannel")
public MessageSource<Integer> numberSource(AtomicInteger source) {
    return () -> {
        return new GenericMessage<>(source.getAndIncrement());
    };
}
```

This @Bean method declares an inbound channel adapter bean which, per the @InboundChannelAdapter annotation, submits a number from the injected AtomicInteger to the channel named numberChannel every 1 second (or 1,000 ms).

Whereas @InboundChannelAdapter indicates an inbound channel adapter when using Java configuration, the from() method is how it's done when using the Java DSL to define the integration flow. The following snippet of a flow definition shows a similar inbound channel adapter as defined in the Java DSL:

```
@Bean
public IntegrationFlow someFlow(AtomicInteger integerSource) {
    return IntegrationFlows
        .from(integerSource, "getAndIncrement",
            c -> c.poller(Pollers.fixedRate(1000)))
        ...
        .get();
}
```

Often, channel adapters are provided by one of Spring Integration's many endpoint modules. Suppose, for example, that you need an inbound channel adapter that monitors a specified directory and submits any files that are written to that directory as messages to a channel named file-channel. The following Java configuration uses FileReadingMessageSource from Spring Integration's file endpoint module to achieve that:

```
@Bean
@InboundChannelAdapter(channel="file-channel",
    poller=@Poller(fixedDelay="1000"))
public MessageSource<File> fileReadingMessageSource() {
    FileReadingMessageSource sourceReader = new FileReadingMessageSource();
    sourceReader.setDirectory(new File(INPUT_DIR));
    sourceReader.setFilter(new SimplePatternFileListFilter(FILE_PATTERN));
    return sourceReader;
}
```

When writing the equivalent file-reading inbound channel adapter in the Java DSL, the inboundAdapter() method from the Files class achieves the same thing. An outbound channel adapter is the end of the line for the integration flow, handing off the final message to the application or to some other system:

```

@Bean
public IntegrationFlow fileReaderFlow() {
    return IntegrationFlows
        .from(Files.inboundAdapter(new File(INPUT_DIR))
            .patternFilter(FILE_PATTERN))
        .get();
}

```

Service activators, implemented as message handlers, often serve the purpose of an outbound channel adapter, especially when data needs to be handed off to the application itself. We've already discussed service activators, so there's no point in repeating that discussion.

It's worth noting, however, that Spring Integration endpoint modules provide useful message handlers for several common use cases. You saw an example of such an outbound channel adapter, `FileWritingMessageHandler`, in listing 9.3. Speaking of Spring Integration endpoint modules, let's take a quick look at what ready-to-use integration endpoint modules are available.

9.2.9 Endpoint modules

It's great that Spring Integration lets you create your own channel adapters. But what's even better is that Spring Integration provides over two dozen endpoint modules containing channel adapters—both inbound and outbound—for integration with a variety of common external systems, including those listed in table 9.1.

Table 9.1 Spring Integration provides over two dozen endpoint models for integration with external systems.

Module	Dependency artifact ID (Group ID: <code>org.springframework.integration</code>)
AMQP	<code>spring-integration-amqp</code>
Spring application events	<code>spring-integration-event</code>
RSS and Atom	<code>spring-integration-feed</code>
Filesystem	<code>spring-integration-file</code>
FTP/FTPS	<code>spring-integration-ftp</code>
GemFire	<code>spring-integration-gemfire</code>
HTTP	<code>spring-integration-http</code>
JDBC	<code>spring-integration-jdbc</code>
JPA	<code>spring-integration-jpa</code>
JMS	<code>spring-integration-jms</code>
Email	<code>spring-integration-mail</code>
MongoDB	<code>spring-integration-mongodb</code>

Table 9.1 Spring Integration provides over two dozen endpoint models for integration with external systems. (continued)

Module	Dependency artifact ID (Group ID: org.springframework.integration)
MQTT	spring-integration-mqtt
Redis	spring-integration-redis
RMI	spring-integration-rmi
SFTP	spring-integration-sftp
STOMP	spring-integration-stomp
Stream	spring-integration-stream
Syslog	spring-integration-syslog
TCP/UDP	spring-integration-ip
Twitter	spring-integration-twitter
Web Services	spring-integration-ws
WebFlux	spring-integration-webflux
WebSocket	spring-integration-websocket
XMPP	spring-integration-xmpp
ZooKeeper	spring-integration-zookeeper

One thing that's clear from looking at table 9.1 is that Spring Integration provides an extensive set of components to meet many integration needs. Most applications will never need even a fraction of what Spring Integration offers. But it's good to know that Spring Integration has you covered if you need them.

What's more, it would be impossible to cover all the channel adapters afforded by the modules listed in table 9.1 in the space of this chapter. You've already seen examples that use the filesystem module to write to the filesystem. And you're soon going to use the email module to read emails.

Each of the endpoint modules offers channel adapters that can be either declared as beans when using Java configuration or referenced via static methods when using Java DSL configuration. I encourage you to explore any of the other endpoint modules that interest you most. You'll find that they're fairly consistent in how they're used. But for now, let's turn our attention to the email endpoint module to see how you might use it in the Taco Cloud application.

9.3 Creating an email integration flow

You've decided that Taco Cloud should enable its customers to submit their taco designs and place orders by email. You send out flyers and place take-out ads in newspapers inviting everyone to send in their taco orders by email. It's a tremendous

success! Unfortunately, it's a bit *too* successful. There are so many emails coming in that you have to hire temporary help to do nothing more than read all the emails and submit order details into the ordering system.

In this section, you'll implement an integration flow that polls the Taco Cloud inbox for taco order emails, parses the emails for order details, and submits the orders to Taco Cloud for handling. In short, the integration flow you're going to need will use an inbound channel adapter from the email endpoint module to ingest emails from the Taco Cloud inbox into the integration flow.

The next step in the integration flow will parse the emails into order objects that are handed off to another handler to submit orders to Taco Cloud's REST API, where they'll be processed the same as any order. To start with, let's define a simple configuration properties class to capture the specifics of how to handle Taco Cloud emails:

```
@Data
@ConfigurationProperties(prefix="tacocloud.email")
@Component
public class EmailProperties {

    private String username;
    private String password;
    private String host;
    private String mailbox;
    private long pollRate = 30000;

    public String getImapUrl() {
        return String.format("imaps://%s:%s@%s/%s",
            this.username, this.password, this.host, this.mailbox);
    }
}
```

As you can see, `EmailProperties` captures properties that are used to produce an IMAP URL. The flow uses this URL to connect to the Taco Cloud email server and poll for emails. Among the properties captured are the email user's username and password, as well as the hostname of the IMAP server, the mailbox to poll, and the rate at which the mailbox is polled (which defaults to every 30 seconds).

The `EmailProperties` class is annotated at the class level with `@ConfigurationProperties` with a `prefix` attribute set to `tacocloud.email`. This means that you can configure the details of consuming an email in the `application.yml` file like this:

```
tacocloud:
  email:
    host: imap.tacocloud.com
    mailbox: INBOX
    username: taco-in-flow
    password: 1L0v3T4c0s
    poll-rate: 10000
```

Now let's use `EmailProperties` to configure the integration flow. The flow you're aiming to create will look a little like figure 9.10.

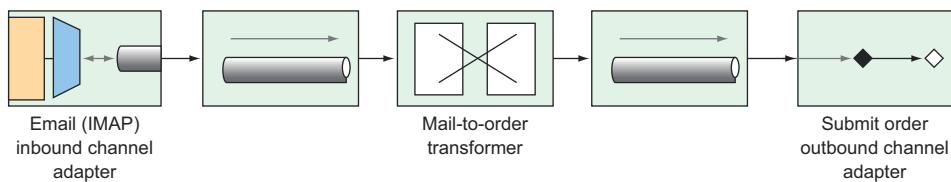


Figure 9.10 An integration flow to accept taco orders by email

You have two options when defining this flow:

- *Define it within the Taco Cloud application itself*—At the end of the flow, a service activator will call into the repositories you've defined to create the taco order.
- *Define it as a separate application*—At the end of the flow, a service activator will send a POST request to the Taco Cloud API to submit the taco order.

Whichever you choose has little bearing on the flow itself, aside from how the service activator is implemented. But because you're going to need some types that represent tacos, orders, and ingredients, which are subtly different than those you've already defined in the main Taco Cloud application, you'll proceed by defining the integration flow in a separate application to avoid any confusion with the existing domain types.

You also have the choice of defining the flow using either XML configuration, Java configuration, or the Java DSL. I rather like the elegance of the Java DSL, so that's what you'll use. Feel free to write the flow using one of the other configuration styles if you're interested in a little extra challenge. For now, let's take a look at the Java DSL configuration for the taco order email flow as shown next.

Listing 9.5 Defining an integration flow to accept emails and submit them as orders

```

package tacos.email;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.Pollers;

@Configuration
public class TacoOrderEmailIntegrationConfig {

    @Bean
    public IntegrationFlow tacoOrderEmailFlow(
        EmailProperties emailProps,
        EmailToOrderTransformer emailToOrderTransformer,
        OrderSubmitMessageHandler orderSubmitHandler) {

        return IntegrationFlows
            .from(Mail imapInboundAdapter(emailProps.getImapUrl()))
            .e -> e.poller(
                Pollers.fixedDelay(emailProps.getPollRate())))
    }
}

```

```

        .transform(emailToOrderTransformer)
        .handle(orderSubmitHandler)
        .get();
    }

}

```

The taco order email flow, as defined in the `tacoOrderEmailFlow()` method, is composed of three distinct components:

- *An IMAP email inbound channel adapter*—This channel adapter is created with the IMP URL generated from the `getImapUrl()` method of `EmailProperties` and polls on a delay set in the `pollRate` property of `EmailProperties`. The emails coming in are handed off to a channel connecting it to the transformer.
- *A transformer that transforms an email into an order object*—The transformer is implemented in `EmailToOrderTransformer`, which is injected into the `tacoOrderEmailFlow()` method. The orders resulting from the transformation are handed off to the final component through another channel.
- *A handler (acting as an outbound channel adapter)*—The handler accepts an order object and submits it to Taco Cloud’s REST API.

The call to `Mail imapInboundAdapter()` is made possible by including the `Email` endpoint module as a dependency in your project build. The Maven dependency looks like this:

```

<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-file</artifactId>
</dependency>

```

The `EmailToOrderTransformer` class is an implementation of Spring Integration’s `Transformer` interface, by way of extending `AbstractMailMessageTransformer` (shown in the following listing).

Listing 9.6 Converting incoming emails to taco orders using an integration transformer

```

@Component
public class EmailToOrderTransformer
    extends AbstractMailMessageTransformer<Order> {

    @Override
    protected AbstractIntegrationMessageBuilder<Order>
        doTransform(Message mailMessage) throws Exception {
        Order tacoOrder = processPayload(mailMessage);
        return MessageBuilder.withPayload(tacoOrder);
    }

    ...
}

```

`AbstractMailMessageTransformer` is a convenient base class for handling messages whose payload is an email. It takes care of extracting the email information from the incoming message into a `Message` object that's passed into the `doTransform()` method.

In the `doTransform()` method, you pass the `Message` to a private method named `processPayload()` to parse the email into an `Order` object. Although similar, the `Order` object in question isn't the same as the `Order` object used in the main Taco Cloud application; it's slightly simpler:

```
package tacos.email;
import java.util.ArrayList;
import java.util.List;
import lombok.Data;

@Data
public class Order {
    private final String email;
    private List<Taco> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {
        this.tacos.add(taco);
    }
}
```

Rather than carry the customer's entire delivery and billing information, this `Order` class only carries the customer's email, obtained from the incoming email.

Parsing emails into taco orders is a non-trivial task. In fact, even a naive implementation involves several dozen lines of code. And those several dozen lines of code do nothing to further the discussion of Spring Integration and how to implement a transformer. Therefore, to save space, I'm leaving out the details of the `processPayload()` method.

The last thing that `EmailToOrderTransformer` does is return a `MessageBuilder` with a payload containing the `Order` object. The message that's produced by the `MessageBuilder` is sent to the final component in the integration flow: a message handler that posts the order to Taco Cloud's API. The `OrderSubmitMessageHandler`, as shown in the next listing, implements Spring Integration's `GenericHandler` to handle messages with an `Order` payload.

Listing 9.7 Posting orders to the Taco Cloud API via a message handler

```
package tacos.email;
import java.util.Map;
import org.springframework.integration.handler.GenericHandler;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class OrderSubmitMessageHandler
    implements GenericHandler<Order> {
```

```

private RestTemplate rest;
private ApiProperties apiProps;

public OrderSubmitMessageHandler(
    ApiProperties apiProps, RestTemplate rest) {
    this.apiProps = apiProps;
    this.rest = rest;
}

@Override
public Object handle(Order order, Map<String, Object> headers) {
    rest.postForObject(apiProps.getUrl(), order, String.class);
    return null;
}
}

```

To satisfy the requirements of the `GenericHandler` interface, `OrderSubmitMessageHandler` overrides the `handle()` method. This method receives the incoming `Order` object and uses an injected `RestTemplate` to submit the `Order` via a POST request to the URL captured in an injected `ApiProperties` object. Finally, the `handle()` method returns `null` to indicate that this handler marks the end of the flow.

`ApiProperties` is used to avoid hardcoding the URL in the call to `postForObject()`. It's a configuration properties file that looks like this:

```

@Data
@ConfigurationProperties(prefix="tacocloud.api")
@Component
public class ApiProperties {
    private String url;
}

```

And in `application.yml`, the URL for the Taco Cloud API might be configured like this:

```

tacocloud:
  api:
    url: http://api.tacocloud.com

```

In order to make `RestTemplate` available in the project so that it can be injected into `OrderSubmitMessageHandler`, you need to add the Spring Boot web starter to the project build:

```

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

```

Whereas this makes `RestTemplate` available in the classpath, it also triggers auto-configuration for Spring MVC. As a standalone Spring Integration flow, the application doesn't need Spring MVC or even the embedded Tomcat that autoconfiguration

provides. Therefore, you should disable Spring MVC autoconfiguration with the following entry in application.yml:

```
spring:  
  main:  
    web-application-type: none
```

The `spring.main.web-application-type` property can be set to either `servlet`, `reactive`, or `none`. When Spring MVC is in the classpath, autoconfiguration sets its value to `servlet`. But here you override it to `none` so that Spring MVC and Tomcat won't be autoconfigured. (We'll talk more about what it means for an application to be a reactive web application in chapter 11.)

Summary

- Spring Integration enables the definition of flows through which data can be processed as it enters or leaves an application.
- Integration flows can be defined in XML, Java, or using a succinct Java DSL configuration style.
- Message gateways and channel adapters act as entry and exit points of an integration flow.
- Messages can be transformed, split, aggregated, routed, and processed by service activators in the course of a flow.
- Message channels connect the components of an integration flow.

Part 3

Reactive Spring

In part 3, we'll explore the exciting new support for reactive programming in Spring. Chapter 10 discusses the essentials of reactive programming with Project Reactor, the reactive programming library that underpins Spring 5's reactive features. We'll then look at some of Reactor's most useful reactive operations. In chapter 11, we'll revisit REST API development, introducing Spring WebFlux, a new web framework that borrows much from Spring MVC while offering a new reactive model for web development. Chapter 12 rounds out part 3 with a look at writing reactive data persistence with Spring Data to read and write data to Cassandra and Mongo databases.

10

Introducing Reactor

This chapter covers

- Understanding reactive programming
- Project Reactor
- Operating on data reactively

Have you ever held a subscription for a newspaper or a magazine? The internet has certainly taken a bite out of the subscriber base of traditional publications, but there was a time when a newspaper subscription was one of the best ways to keep up with the events of the day. You could count on a fresh delivery of current events every morning, to read during breakfast or on the way to work.

Now suppose that if, after paying for your subscription, several days go by and no papers have been delivered. A few more days go by, and you call the newspaper sales office to ask why you haven't yet received your daily paper. Imagine your surprise if they explain, "You paid for a full year of newspapers. The year hasn't completed yet. You'll certainly receive them all once the full year of newspapers is ready."

Thankfully, that's not at all how subscriptions work. Newspapers have a certain timeliness to them. They're delivered as quickly as possible after publication so that they can be read while their content is still fresh. Moreover, as you're reading the

latest issue, newspaper reporters are writing new stories for future editions, and the presses are fired up producing the next edition—all in parallel.

As we develop application code, there are two styles of code we can write: imperative and reactive:

- *Imperative* code is a lot like that absurd hypothetical newspaper subscription. It's a serial set of tasks, each running one at a time, each after the previous task. Data is processed in bulk and can't be handed over to the next task until the previous task has completed its work on the bulk of data.
- *Reactive* code is a lot like a real newspaper subscription. A set of tasks is defined to process data, but those tasks can run in parallel. Each task can process subsets of the data, handing it off to the next task in line while it continues to work on another subset of the data.

In this chapter, we're going to step away from the Taco Cloud application temporarily to explore Project Reactor. Reactor is a library for reactive programming that's part of the Spring family of projects. And because it serves as the foundation of Spring 5's support for reactive programming, it's important that you understand Reactor before we look at building reactive controllers and repositories with Spring. Before we start working with Reactor, though, let's quickly examine the essentials of reactive programming.

10.1 **Understanding reactive programming**

Reactive programming is a paradigm that's an alternative to imperative programming. This alternative exists because reactive programming addresses a limitation in imperative programming. By understanding these limitations, you can better grasp the benefits of the reactive model.

NOTE Reactive programming isn't a silver bullet. In no way should you infer from this chapter or any other discussion of reactive programming that imperative programming is evil and that reactive programming is your savior. Like anything you learn as a developer, reactive programming is a perfect fit in some use cases, and it's ill fitted in others. An ounce of pragmatism is advised.

If you're like me and many developers, you cut your programming teeth with imperative programming. There's a good chance that most (or all) of the code you write today is still imperative in nature. Imperative programming is intuitive enough that young students are learning it with ease in their school's STEM programs, and it's powerful enough that it makes up the bulk of code that drives the largest enterprises.

The idea is simple: you write code as a list of instructions to be followed, one at a time, in the order that they're encountered. A task is performed and the program waits for it to complete before moving on to the next task. At each step along the way, the data that's to be processed must be fully available so that it can be processed as a whole.

This is fine ... until it isn't. While a task is being performed, and especially if it's an I/O task such as writing data to a database or fetching data from a remote server, the thread that invoked that task is blocked, unable to do anything else until the task completes. To put it bluntly, blocked threads are wasteful.

Most programming languages, including Java, support concurrent programming. It's fairly easy to fire up another thread in Java and send it on its way to perform some work while the invoking thread carries on with something else. But although it's easy to create threads, those threads are likely to end up blocked themselves. Managing concurrency in multiple threads is challenging. More threads mean more complexity.

In contrast, reactive programming is functional and declarative in nature. Rather than describe a set of steps that are to be performed sequentially, reactive programming involves describing a pipeline or stream through which data flows. Rather than requiring the data be available to be processed as a whole, a reactive stream processes data as it becomes available. In fact, the incoming data may be endless (a constant stream of a location's real-time temperature data, for instance).

To apply a real-world analogy, consider imperative programming as a water balloon and reactive programming as a garden hose. Both are suitable ways to surprise and soak an unsuspecting friend on a hot summer day. But they differ in their execution style:

- A water balloon carries its payload all at once, soaking its intended target at the moment of impact. The water balloon has a finite capacity, however, and if you wish to soak more people (or the same person to a greater extent), your only choice is to scale up by increasing the number of water balloons.
- A garden hose carries its payload as a stream of water that flows from the spigot to the nozzle. The garden hose's capacity may be finite at any given point in time, but it's unlimited over the course of a water battle. As long as water is entering the hose from the spigot, it will continue to flow through the hose and spray out of the nozzle. The same garden hose is easily scalable to soak as many friends as you wish.

There's nothing inherently wrong with water balloons (or imperative programming), but the person holding the garden hose (or applying reactive programming) has an advantage in regard to scalability and performance.

10.1.1 Defining Reactive Streams

Reactive Streams is an initiative started in late 2013 by engineers from Netflix, Lightbend, and Pivotal (the company behind Spring). Reactive Streams aims to provide a standard for asynchronous stream processing with non-blocking backpressure.

We've already touched on the asynchronous trait of reactive programming; it's what enables us to perform tasks in parallel to achieve greater scalability. Backpressure is a means by which consumers of data can avoid being overwhelmed by an overly fast data source, by establishing limits on how much they're willing to handle.

Java Streams vs. Reactive Streams

There's a lot of similarity between Java Streams and Reactive Streams. To start with, they both have the word *Streams* in their names. They also both provide a functional API for working with data. In fact, as you'll see later when we look at Reactor, they even share many of the same operations.

Java Streams, however, are typically synchronous and work with a finite set of data. They're essentially a means of iterating over a collection with functions.

Reactive Streams support asynchronous processing of datasets of any size, including infinite datasets. They process data in real time, as it becomes available, with back-pressure to avoid overwhelming their consumers.

The Reactive Streams specification can be summed up by four interface definitions: Publisher, Subscriber, Subscription, and Processor. A Publisher produces data that it sends to a Subscriber per a Subscription. The Publisher interface declares a single method, `subscribe()`, through which a Subscriber can subscribe to the Publisher:

```
public interface Publisher<T> {  
    void subscribe(Subscriber<? super T> subscriber);  
}
```

Once a Subscriber has subscribed, it can receive events from the Publisher. Those events are sent via methods on the Subscriber interface:

```
public interface Subscriber<T> {  
    void onSubscribe(Subscription sub);  
    void onNext(T item);  
    void onError(Throwable ex);  
    void onComplete();  
}
```

The first event that the Subscriber will receive is through a call to `onSubscribe()`. When the Publisher calls `onSubscribe()`, it passes a `Subscription` object to the Subscriber. It's through the `Subscription` that the Subscriber can manage its subscription:

```
public interface Subscription {  
    void request(long n);  
    void cancel();  
}
```

The Subscriber can call `request()` to request that data be sent, or it can call `cancel()` to indicate that it's no longer interested in receiving data and is canceling the subscription. When calling `request()`, the Subscriber passes in a `long` value to indicate how many data items it's willing to accept. This is where backpressure comes in, preventing the Publisher from sending more data than the Subscriber is able to handle. After the Publisher has sent as many items as were requested, the Subscriber can call `request()` again to request more.

Once the Subscriber has requested data, the data starts flowing through the stream. For every item that's published by the Publisher, the `onNext()` method will be called to deliver the data to the Subscriber. If there are any errors, `onError()` is called. If the Publisher has no more data to send and isn't going to produce any more data, it will call `onComplete()` to tell the Subscriber that it's out of business.

As for the Processor interface, it's a combination of Subscriber and Publisher, as shown here:

```
public interface Processor<T, R>
    extends Subscriber<T>, Publisher<R> {}
```

As a Subscriber, a Processor will receive data and process it in some way. Then it will switch hats and act as a Publisher to publish the results to its Subscribers.

As you can see, the Reactive Streams specification is rather straightforward. It's fairly easy to see how you could build up a data-processing pipeline that starts with a Publisher, pumps data through zero or more Processors, and then drops the final results off to a Subscriber.

What the Reactive Streams interfaces don't lend themselves to, however, is composing such a stream in a functional way. Project Reactor is an implementation of the Reactive Streams specification that provides a functional API for composing Reactive Streams. As you'll see in the following chapters, Reactor is the foundation for Spring 5's reactive programming model. In the remainder of this chapter, we're going to explore (and, dare I say, have a lot of fun with) Project Reactor.

10.2 Getting started with Reactor

Reactive programming requires us to think in a very different way from imperative programming. Rather than describe a set of steps to be taken, reactive programming means building a pipeline through which data will flow. As data passes through the pipeline, it can be altered or used in some way.

For example, suppose you want to take a person's name, change all of its letters to uppercase, use it to create a greeting message, and then finally print it. In an imperative programming model, the code would look something like this:

```
String name = "Craig";
String capitalName = name.toUpperCase();
String greeting = "Hello, " + capitalName + "!";
System.out.println(greeting);
```

In the imperative model, each line of code performs a step, one right after the other, and definitely in the same thread. Each step blocks the executing thread from moving to the next step until complete.

In contrast, functional, reactive code could achieve the same thing like this:

```
Mono.just("Craig")
    .map(n -> n.toUpperCase())
    .map(cn -> "Hello, " + cn + "!")
    .subscribe(System.out::println);
```

Don't worry too much about the details of this example; we'll talk all about the `just()`, `map()`, and `subscribe()` operations soon enough. For now, it's important to understand that although the reactive example still seems to follow a step-by-step model, it's really a pipeline that data flows through. At each phase of the pipeline, the data is tweaked somehow, but no assumption can be made about which thread any of the operations are performed on. They may be the same thread ... or they may not be.

The `Mono` in the example is one of Reactor's two core types. `Flux` is the other. Both are implementations of Reactive Streams' `Publisher`. A `Flux` represents a pipeline of zero, one, or many (potentially infinite) data items. A `Mono` is a specialized reactive type that's optimized for when the dataset is known to have no more than one data item.

Reactor vs. RxJava (ReactiveX)

If you're already familiar with RxJava or ReactiveX, you may be thinking that `Mono` and `Flux` sound a lot like `Observable` and `Single`. In fact, they're approximately equivalent semantically. They even offer many of the same operations.

Although we focus on Reactor in this book, you may be happy to know that it's possible to convert between Reactor and RxJava types. Moreover, as you'll see in the following chapters, Spring can also work with RxJava types.

There are actually three `Mono`s in the previous example. The `just()` operation creates the first one. When the `Mono` emits a value, that value is given to the `map()` operation to be capitalized and used to create another `Mono`. When the second `Mono` publishes its data, it's given to the second `map()` operation to do some `String` concatenation, the results of which are used to create the third `Mono`. Finally, the call to `subscribe()` subscribes to the `Mono`, receives the data, and prints it.

10.2.1 Diagramming reactive flows

Reactive flows are often illustrated with marble diagrams. Marble diagrams, in their simplest form, depict a timeline of data as it flows through a `Flux` or `Mono` at the top, an operation in the middle, and the timeline of the resulting `Flux` or `Mono` at the bottom. Figure 10.1 shows a marble diagram template for a `Flux`. As you can see, as data flows through the original `Flux`, it's processed through some operation, resulting in a new `Flux` through which the processed data flows.

Figure 10.2 shows a similar marble diagram, but for a `Mono`. As you can see, the key difference is that a `Mono` will have either zero or one data item, or an error.

In section 10.3, we'll explore many operations supported by `Flux` and `Mono`, and we'll use marble diagrams to visualize how they work.

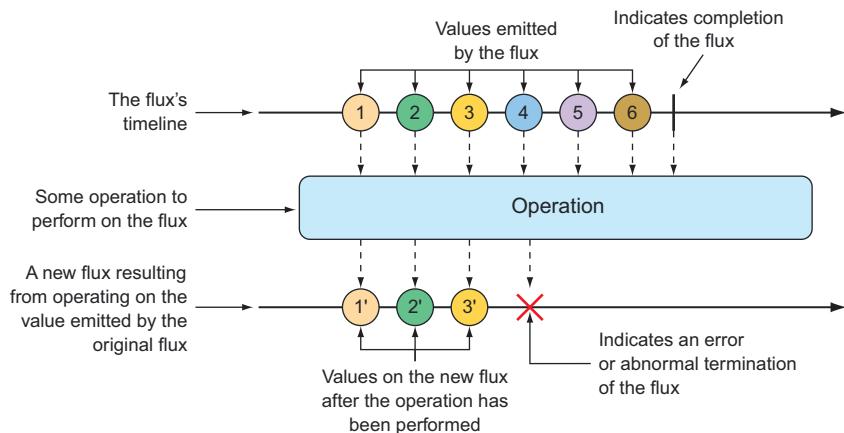


Figure 10.1 Marble diagram illustrating the basic flow of a Flux

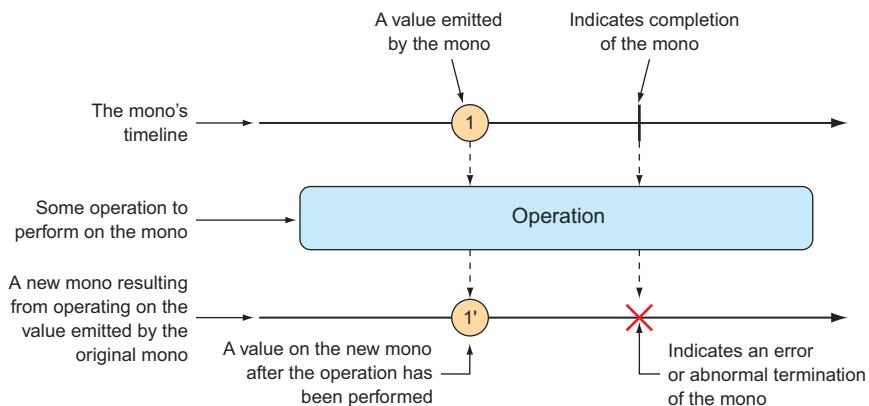


Figure 10.2 Marble diagram illustrating the basic flow of a Mono

10.2.2 Adding Reactor dependencies

To get started with Reactor, add the following dependency to the project build:

```
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-core</artifactId>
</dependency>
```

Reactor also provides some great testing support. You're going to write a lot of tests around your Reactor code, so you'll definitely want to add this dependency to your build:

```
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-test</artifactId>
</dependency>
```

```
<scope>test</scope>
</dependency>
```

I'm assuming that you're adding these dependencies to a Spring Boot project, which handles dependency management for you, so there's no need to specify the `<version>` element for the dependencies. But if you want to use Reactor in a non-Spring Boot project, you'll need to set up Reactor's BOM (bill of materials) in the build. The following dependency management entry adds Reactor's Bismuth release to the build:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.projectreactor</groupId>
      <artifactId>reactor-bom</artifactId>
      <version>Bismuth-RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Now that Reactor is in your project build, you can start creating reactive pipelines with Mono and Flux. For the remainder of this chapter, we'll walk through several operations offered by Mono and Flux.

10.3 Applying common reactive operations

Flux and Mono are the most essential building blocks provided by Reactor, and the operations those two reactive types offer are the mortar that binds them together to create pipelines through which data can flow. Between Flux and Mono, there are over 500 operations, each of which can be loosely categorized as

- Creation operations
- Combination operations
- Transformation operations
- Logic operations

As much fun as it would be to poke at each of the 500 operations to see how they tick, there's simply not enough room in this chapter. I've selected a few of the most useful operations to experiment with in this section. We'll start with creation operations.

NOTE Where are the Mono examples? Mono and Flux share many of the same operations, so it's mostly unnecessary to show the same operation twice, once for Mono and again for Flux. Moreover, although the Mono operations are useful, they're slightly less interesting to look at than the same operations when given a Flux. Most of the examples we'll work with will involve Flux. Just know that Mono often has equivalent operations.

10.3.1 Creating reactive types

Often when working with reactive types in Spring, you'll be given a Flux or a Mono from a repository or a service, so you won't need to create one yourself. But occasionally you'll need to create a new reactive publisher.

Reactor provides several operations for creating Fluxes and Monos. In this section, we'll look at a few of the most useful creation operations.

CREATING FROM OBJECTS

If you have one or more objects that you'd like to create a Flux or Mono from, you can use the static `just()` method on Flux or Mono to create a reactive type whose data is driven by those objects. For example, the following test method creates a Flux from five String objects:

```
@Test
public void createAFlux_just() {
    Flux<String> fruitFlux = Flux
        .just("Apple", "Orange", "Grape", "Banana", "Strawberry");
}
```

At this point, the Flux has been created, but it has no subscribers. Without any subscribers, data won't flow. Thinking of the garden hose analogy, you've attached the garden hose to the spigot, and there's water from the utility company on the other side—but until you turn on the spigot, water won't flow. Subscribing to a reactive type is how you turn on the flow of data.

To add a subscriber, you can call the `subscribe()` method on the Flux:

```
fruitFlux.subscribe(
    f -> System.out.println("Here's some fruit: " + f)
);
```

The lambda given to `subscribe()` here is actually a `java.util.Consumer` that's used to create a Reactive Streams Subscriber. Upon calling `subscribe()`, the data starts flowing. In this example, there are no intermediate operations, so the data flows directly from the Flux to the Subscriber.

Printing the entries from a Flux or Mono to the console is a good way to see the reactive type in action. But a better way to actually test a Flux or a Mono is to use Reactor's `StepVerifier`. Given a Flux or Mono, `StepVerifier` subscribes to the reactive type and then applies assertions against the data as it flows through the stream, finally verifying that the stream completes as expected.

For example, to verify that the prescribed data flows through the `fruitFlux`, you can write a test that looks like this:

```
StepVerifier.create(fruitFlux)
    .expectNext("Apple")
    .expectNext("Orange")
    .expectNext("Grape")
```

```
.expectNext("Banana")
.expectNext("Strawberry")
.verifyComplete();
```

In this case, StepVerifier subscribes to the Flux and then asserts that each item matches the expected fruit name. Finally, it verifies that after Strawberry is produced by the Flux, the Flux is complete.

For the remainder of the examples in this chapter, you'll use StepVerifier to write learning tests—tests that verify behavior and help you understand how something works—to get to know some of Reactor's most useful operations.

CREATING FROM COLLECTIONS

A Flux can also be created from an array, Iterable, or Java Stream. Figure 10.3 illustrates how this works with a marble diagram.

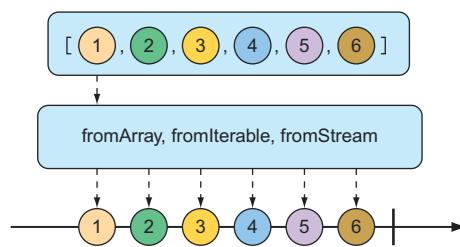


Figure 10.3 A Flux can be created from an array, Iterable, or Stream.

To create a Flux from an array, call the static `fromArray()` method, passing in the source array:

```
@Test
public void createAFlux_fromArray() {
    String[] fruits = new String[] {
        "Apple", "Orange", "Grape", "Banana", "Strawberry" };

    Flux<String> fruitFlux = Flux.fromArray(fruits);

    StepVerifier.create(fruitFlux)
        .expectNext("Apple")
        .expectNext("Orange")
        .expectNext("Grape")
        .expectNext("Banana")
        .expectNext("Strawberry")
        .verifyComplete();
}
```

Because the source array contains the same fruit names you used when creating a Flux from a list of objects, the data emitted by the Flux will have the same values. Thus, you can use the same StepVerifier as before to verify this Flux.

If you need to create a Flux from a `java.util.List`, `java.util.Set`, or any other implementation of `java.lang.Iterable`, you can pass it into the static `fromIterable()` method:

```
@Test
public void createAFlux_fromIterable() {
    List<String> fruitList = new ArrayList<>();
    fruitList.add("Apple");
    fruitList.add("Orange");
    fruitList.add("Grape");
    fruitList.add("Banana");
    fruitList.add("Strawberry");

    Flux<String> fruitFlux = Flux.fromIterable(fruitList);

    // ... verify steps
}
```

Or, if you happen to have a Java Stream that you'd like to use as the source for a Flux, `fromStream()` is the method you'll use:

```
@Test
public void createAFlux_fromStream() {
    Stream<String> fruitStream =
        Stream.of("Apple", "Orange", "Grape", "Banana", "Strawberry");

    Flux<String> fruitFlux = Flux.fromStream(fruitStream);

    // ... verify steps
}
```

Again, the same `StepVerifier` as before can be used to verify the data published by the Flux.

GENERATING FLUX DATA

Sometimes you don't have any data to work with and just need Flux to act as a counter, emitting a number that increments with each new value. To create a counter Flux, you can use the static `range()` method. The diagram in figure 10.4 illustrates how `range()` works.

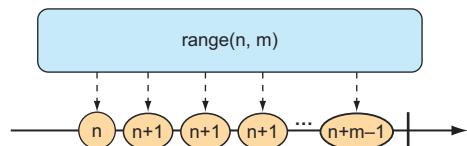


Figure 10.4 Creating a Flux from a range results in a counter-style publishing of messages.

The following test method demonstrates how to create a range Flux:

```
@Test
public void createAFlux_range() {
    Flux<Integer> intervalFlux =
        Flux.range(1, 5);
```

```

StepVerifier.create(intervalFlux)
    .expectNext(1)
    .expectNext(2)
    .expectNext(3)
    .expectNext(4)
    .expectNext(5)
    .verifyComplete();
}

```

In this example, the range Flux is created with a starting value of 1 and an ending value of 5. The StepVerifier proves that it will publish five items, which are the integers 1 through 5.

Another Flux-creation method that's similar to `range()` is `interval()`. Like the `range()` method, `interval()` creates a Flux that emits an incrementing value. But what makes `interval()` special is that instead of you giving it a starting and ending value, you specify a duration or how often a value should be emitted. Figure 10.5 shows a marble diagram for the `interval()` creation method.

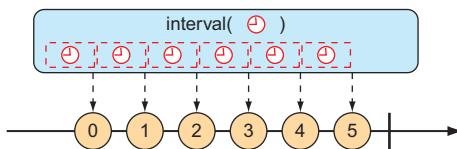


Figure 10.5 A Flux created from an interval has a periodic entry published to it.

For example, to create an interval Flux that emits a value every second, you can use the static `interval()` method as follows:

```

@Test
public void createAFlux_interval() {
    Flux<Long> intervalFlux =
        Flux.interval(Duration.ofSeconds(1))
            .take(5);

    StepVerifier.create(intervalFlux)
        .expectNext(0L)
        .expectNext(1L)
        .expectNext(2L)
        .expectNext(3L)
        .expectNext(4L)
        .verifyComplete();
}

```

Notice that the value emitted by an interval Flux starts with 0 and increments on each successive item. Also, because `interval()` isn't given a maximum value, it will potentially run forever. Therefore, you also use the `take()` operation to limit the results to the first five entries. We'll talk more about the `take()` operation in the next section.

10.3.2 Combining reactive types

You may find yourself with two reactive types that you need to somehow merge together. Or, in other cases, you may need to split a Flux into more than one reactive type. In this section, we'll examine operations that combine and split Reactor's Flux and Mono.

MERGING REACTIVE TYPES

Suppose you have two Flux streams and need to create a single resulting Flux that will produce data as it becomes available from either of the upstream Flux streams. To merge one Flux with another, you can use the `mergeWith()` operation, as illustrated with the marble diagram in figure 10.6.

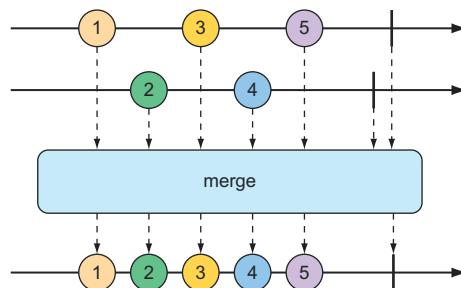


Figure 10.6 Merging two Flux streams interleaves their messages into a new Flux.

For example, suppose you have a Flux whose values are the names of TV and movie characters, and you have a second Flux whose values are the names of foods that those characters enjoy eating. The following test method shows how you could merge the two Flux objects with the `mergeWith()` method:

```

@Test
public void mergeFluxes() {
    Flux<String> characterFlux = Flux
        .just("Garfield", "Kojak", "Barbossa")
        .delayElements(Duration.ofMillis(500));
    Flux<String> foodFlux = Flux
        .just("Lasagna", "Lollipops", "Apples")
        .delaySubscription(Duration.ofMillis(250))
        .delayElements(Duration.ofMillis(500));

    Flux<String> mergedFlux = characterFlux.mergeWith(foodFlux);

    StepVerifier.create(mergedFlux)
        .expectNext("Garfield")
        .expectNext("Lasagna")
        .expectNext("Kojak")
        .expectNext("Lollipops")
        .expectNext("Barbossa")
        .expectNext("Apples")
        .verifyComplete();
}
  
```

Normally, a Flux will publish data as quickly as it possibly can. Therefore, you use a `delayElements()` operation on both of the created Flux streams to slow them down a little—only emitting an entry every 500 ms. Furthermore, so that the food Flux starts streaming after the character Flux, you apply a `delaySubscription()` operation to the food Flux so that it won’t emit any data until 250 ms have passed following a subscription.

After merging the two Flux objects, a new merged Flux is created. When `StepVerifier` subscribes to the merged Flux, it will, in turn, subscribe to the two source Flux streams, starting the flow of data.

The order of items emitted from the merged Flux aligns with the timing of how they’re emitted from the sources. Because both Flux objects are set to emit at regular rates, the values will be interleaved through the merged Flux, resulting in a character, followed by a food, followed by a character, and so forth. If the timing of either Flux were to change, it’s possible that you might see two character items or two food items published one after the other.

Because `mergeWith()` can’t guarantee a perfect back and forth between its sources, you may want to consider the `zip()` operation instead. When two Flux objects are zipped together, it results in a new Flux that produces a tuple of items, where the tuple contains one item from each source Flux. Figure 10.7 illustrates how two Flux objects can be zipped together.

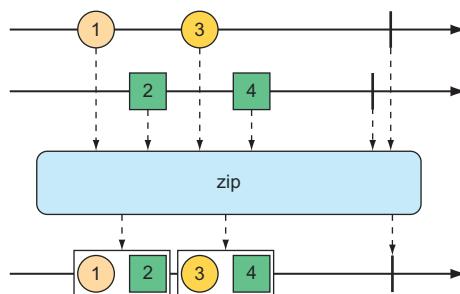


Figure 10.7 Zipping two Flux streams results in a Flux containing tuples of one element from each Flux.

To see the `zip()` operation in action, consider the following test method, which zips the character Flux and the food Flux together:

```

@Test
public void zipFluxes() {
    Flux<String> characterFlux = Flux
        .just("Garfield", "Kojak", "Barbossa");
    Flux<String> foodFlux = Flux
        .just("Lasagna", "Lollipops", "Apples");

    Flux<Tuple2<String, String>> zippedFlux =
        Flux.zip(characterFlux, foodFlux);
  
```

```

StepVerifier.create(zippedFlux)
    .expectNextMatches(p ->
        p.getT1().equals("Garfield") &&
        p.getT2().equals("Lasagna"))
    .expectNextMatches(p ->
        p.getT1().equals("Kojak") &&
        p.getT2().equals("Lollipops"))
    .expectNextMatches(p ->
        p.getT1().equals("Barbossa") &&
        p.getT2().equals("Apples"))
    .verifyComplete();
}

```

Notice that unlike `mergeWith()`, the `zip()` operation is a static creation operation. The created Flux has a perfect alignment between characters and their favorite foods. Each item emitted from the zipped Flux is a `Tuple2` (a container object that carries two other objects) containing items from each source Flux, in the order that they're published.

If you'd rather not work with a `Tuple2` and would rather work with some other type, you can provide a Function to `zip()` that produces any object you'd like, given the two items (as shown in the marble diagram in figure 10.8).

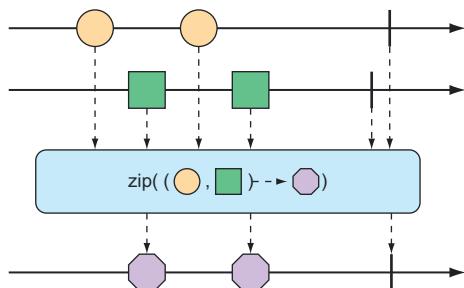


Figure 10.8 An alternative form of the `zip` operation results in a Flux of messages created from one element of each incoming Flux.

For example, the following test method shows how to zip the character Flux with the food Flux so that it results in a Flux of String objects:

```

@Test
public void zipFluxesToObject() {
    Flux<String> characterFlux = Flux
        .just("Garfield", "Kojak", "Barbossa");
    Flux<String> foodFlux = Flux
        .just("Lasagna", "Lollipops", "Apples");

    Flux<String> zippedFlux =
        Flux.zip(characterFlux, foodFlux, (c, f) -> c + " eats " + f);

    StepVerifier.create(zippedFlux)
        .expectNext("Garfield eats Lasagna")
        .expectNext("Kojak eats Lollipops")
        .expectNext("Barbossa eats Apples")
        .verifyComplete();
}

```

The Function given to `zip()` (given here as a lambda) simply concatenates the two items into a sentence to be emitted by the zipped Flux.

SELECTING THE FIRST REACTIVE TYPE TO PUBLISH

Suppose you have two Flux objects, and rather than merge them together, you merely want to create a new Flux that emits the values from the first Flux that produces a value. As shown in figure 10.9, the `first()` operation picks the first of two Flux objects and echoes the values it publishes.

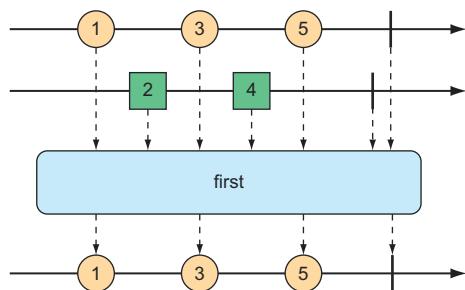


Figure 10.9 The `first` operation chooses the first Flux to emit a message and thereafter only produces messages from that Flux.

The following test method creates a fast Flux and a slow Flux (where “slow” means that it will not publish an item until 100 ms after subscription). Using `first()`, it creates a new Flux that will only publish values from the first source Flux to publish a value:

```

@Test
public void firstFlux() {
    Flux<String> slowFlux = Flux.just("tortoise", "snail", "sloth")
        .delaySubscription(Duration.ofMillis(100));
    Flux<String> fastFlux = Flux.just("hare", "cheetah", "squirrel");

    Flux<String> firstFlux = Flux.first(slowFlux, fastFlux);

    StepVerifier.create(firstFlux)
        .expectNext("hare")
        .expectNext("cheetah")
        .expectNext("squirrel")
        .verifyComplete();
}
  
```

In this case, because the slow Flux won’t publish any values until 100 ms after the fast Flux has started publishing, the newly created Flux will simply ignore the slow Flux and only publish values from the fast Flux.

10.3.3 Transforming and filtering reactive streams

As data flows through a stream, you'll likely need to filter out some values and modify other values. In this section, we'll look at operations that transform and filter the data flowing through a reactive stream.

FILTERING DATA FROM REACTIVE TYPES

One of the most basic ways of filtering data as it flows from a Flux is to simply disregard the first so many entries. The `skip()` operation, illustrated in figure 10.10, does exactly that.

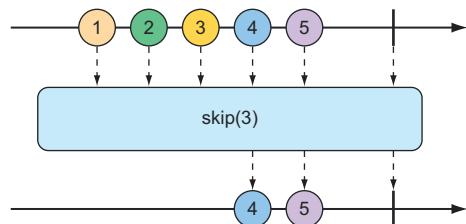


Figure 10.10 The `skip` operation skips a specified number of messages before passing the remaining messages on to the resulting Flux.

Given a Flux with several entries, the `skip()` operation will create a new Flux that skips over a specified number of items before emitting the remaining items from the source Flux. The following test method shows how to use `skip()`:

```
@Test
public void skipAFew() {
    Flux<String> skipFlux = Flux.just(
        "one", "two", "skip a few", "ninety nine", "one hundred")
        .skip(3);

    StepVerifier.create(skipFlux)
        .expectNext("ninety nine", "one hundred")
        .verifyComplete();
}
```

In this case, you have a Flux of five String items. Calling `skip(3)` on that Flux produces a new Flux that skips over the first three items, and only publishes the last two items.

But maybe you don't want to skip a specific number of items, but instead need to skip the first so many items until some duration has passed. An alternate form of the `skip()` operation, illustrated in figure 10.11, produces a Flux that waits until some specified time has passed before emitting items from the source Flux.

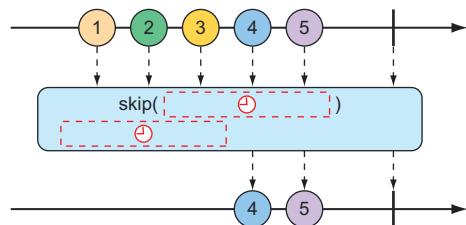


Figure 10.11 An alternative form of the `skip` operation waits until some duration has passed before passing messages on to the resulting Flux.

The test method that follows uses `skip()` to create a Flux that waits four seconds before emitting any values. Because that Flux was created from a Flux that has a one-second delay between items (using `delayElements()`), only the last two items will be emitted:

```
@Test
public void skipAFewSeconds() {
    Flux<String> skipFlux = Flux.just(
        "one", "two", "skip a few", "ninety nine", "one hundred")
        .delayElements(Duration.ofSeconds(1))
        .skip(Duration.ofSeconds(4));

    StepVerifier.create(skipFlux)
        .expectNext("ninety nine", "one hundred")
        .verifyComplete();
}
```

You've already seen an example of the `take()` operation, but in light of the `skip()` operation, `take()` can be thought of as the opposite of `skip()`. Whereas `skip()` skips the first few items, `take()` only emits the first so many items (as illustrated by the marble diagram in figure 10.12):

```
@Test
public void take() {
    Flux<String> nationalParkFlux = Flux.just(
        "Yellowstone", "Yosemite", "Grand Canyon",
        "Zion", "Grand Teton")
        .take(3);

    StepVerifier.create(nationalParkFlux)
        .expectNext("Yellowstone", "Yosemite", "Grand Canyon")
        .verifyComplete();
}
```

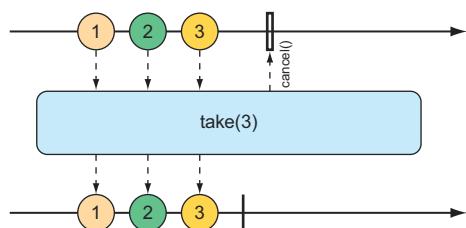


Figure 10.12 The `take` operation passes only the first so many messages from the incoming Flux and then cancels the subscription.

Like `skip()`, `take()` also has an alternative form that's based on a duration rather than an item count. It will take and emit as many items as pass through the source Flux until some period of time has passed, after which the Flux completes. This is illustrated in figure 10.13.

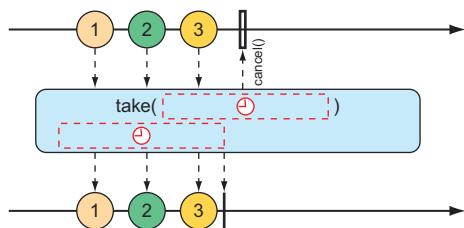


Figure 10.13 An alternative form of the take operation passes messages on to the resulting Flux until some duration has passed.

The following test method uses the alternative form of take() to emit as many items as it can in the first 3.5 seconds after subscription:

```
@Test
public void take() {
    Flux<String> nationalParkFlux = Flux.just(
        "Yellowstone", "Yosemite", "Grand Canyon",
        "Zion", "Grand Teton")
        .delayElements(Duration.ofSeconds(1))
        .take(Duration.ofMillis(3500));

    StepVerifier.create(nationalParkFlux)
        .expectNext("Yellowstone", "Yosemite", "Grand Canyon")
        .verifyComplete();
}
```

The skip() and take() operations can be thought of as filter operations where the filter criteria are based on a count or a duration. For more general-purpose filtering of Flux values, you'll find the filter() operation quite useful.

Given a Predicate that decides whether an item will pass through the Flux or not, the filter() operation lets you selectively publish based on whatever criteria you want. The marble diagram in figure 10.14 shows how filter() works.

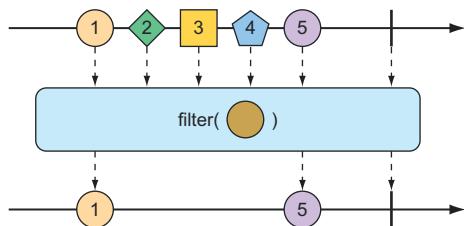


Figure 10.14 An incoming Flux can be filtered so that the resulting Flux only receives messages that match a given predicate.

To see filter() in action, consider the following test method:

```
@Test
public void filter() {
    Flux<String> nationalParkFlux = Flux.just(
        "Yellowstone", "Yosemite", "Grand Canyon",
        "Zion", "Grand Teton")
        .filter(np -> !np.contains(" "));
```

```
StepVerifier.create(nationalParkFlux)
    .expectNext("Yellowstone", "Yosemite", "Zion")
    .verifyComplete();
}
```

Here, `filter()` is given a Predicate as a lambda that only accepts String values that don't have any spaces. Consequently, "Grand Canyon" and "Grand Teton" are filtered out of the resulting Flux.

Perhaps the filtering you need is to filter out any items that you've already received. The `distinct()` operation, as illustrated in figure 10.15, results in a Flux that only publishes items from the source Flux that haven't already been published.

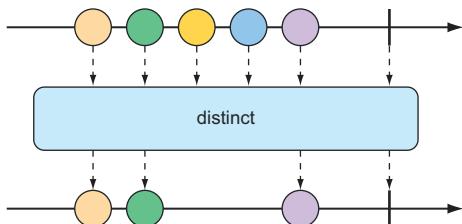


Figure 10.15 The `distinct` operation filters out any duplicate messages.

In the following test, only unique String values will be emitted from the `distinct` Flux:

```
@Test
public void distinct() {
    Flux<String> animalFlux = Flux.just(
        "dog", "cat", "bird", "dog", "bird", "anteater")
        .distinct();

    StepVerifier.create(animalFlux)
        .expectNext("dog", "cat", "bird", "anteater")
        .verifyComplete();
}
```

Although "dog" and "bird" are each published twice from the source Flux, the `distinct` Flux only publishes them once.

MAPPING REACTIVE DATA

One of the most common operations you'll use on either a Flux or a Mono is to transform published items to some other form or type. Reactor's types offer `map()` and `flatMap()` operations for that purpose.

The `map()` operation creates a Flux that simply performs a transformation as prescribed by a given Function on each object it receives before republishing it. Figure 10.16 illustrates how the `map()` operation works.

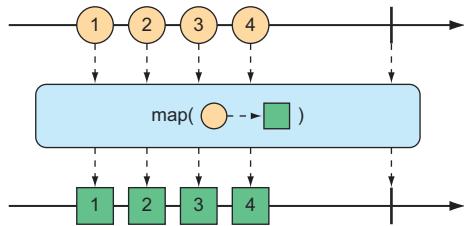


Figure 10.16 The map operation performs a transformation of incoming messages into new messages on the resulting stream.

In the following test method, a Flux of String values representing basketball players is mapped to a new Flux of Player objects:

```
@Test
public void map() {
    Flux<Player> playerFlux = Flux
        .just("Michael Jordan", "Scottie Pippen", "Steve Kerr")
        .map(n -> {
            String[] split = n.split("\\s");
            return new Player(split[0], split[1]);
        });

    StepVerifier.create(playerFlux)
        .expectNext(new Player("Michael", "Jordan"))
        .expectNext(new Player("Scottie", "Pippen"))
        .expectNext(new Player("Steve", "Kerr"))
        .verifyComplete();
}
```

The Function given to `map()` (as a lambda) splits the incoming String at a space and uses the resulting String array to create a Player object. Although the Flux created with `just()` carried String objects, the Flux resulting from `map()` carries Player objects.

What's important to understand about `map()` is that the mapping is performed synchronously, as each item is published by the source Flux. If you want to perform the mapping asynchronously, you should consider the `flatMap()` operation.

The `flatMap()` operation requires some thought and practice to acquire full proficiency. As shown in figure 10.17, instead of simply mapping one object to another, as in the case of `map()`, `flatMap()` maps each object to a new Mono or Flux. The results of the Mono or Flux are flattened into a new resulting Flux. When used along with `subscribeOn()`, `flatMap()` can unleash the asynchronous power of Reactor's types.

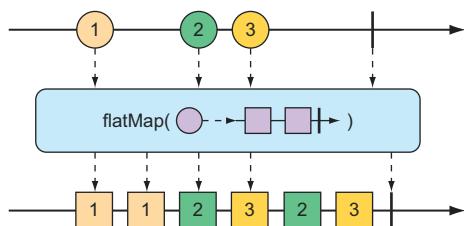


Figure 10.17 The flat map operation uses an intermediate Flux to perform a transformation, consequently allowing for asynchronous transformations.

The following test method demonstrates the use of `flatMap()` and `subscribeOn()`:

```

@Test
public void flatMap() {
    Flux<Player> playerFlux = Flux
        .just("Michael Jordan", "Scottie Pippen", "Steve Kerr")
        .flatMap(n -> Mono.just(n)
            .map(p -> {
                String[] split = p.split("\\s");
                return new Player(split[0], split[1]);
            })
        .subscribeOn(Schedulers.parallel())
    );

    List<Player> playerList = Arrays.asList(
        new Player("Michael", "Jordan"),
        new Player("Scottie", "Pippen"),
        new Player("Steve", "Kerr"));
}

StepVerifier.create(playerFlux)
    .expectNextMatches(p -> playerList.contains(p))
    .expectNextMatches(p -> playerList.contains(p))
    .expectNextMatches(p -> playerList.contains(p))
    .verifyComplete();
}

```

Notice that `flatMap()` is given a lambda function that transforms the incoming `String` into a `Mono` of type `String`. A `map()` operation is then applied to the `Mono` to transform the `String` to a `Player`.

If you stopped right there, the resulting `Flux` would carry `Player` objects, produced synchronously in the same order as with the `map()` example. But the last thing you do with the `Mono` is call `subscribeOn()` to indicate that each subscription should take place in a parallel thread. Consequently, the mapping operations for multiple incoming `String` objects can be performed asynchronously and in parallel.

Although `subscribeOn()` is named similarly to `subscribe()`, they're quite different. Whereas `subscribe()` is a verb, subscribing to a reactive flow and effectively kicking it off, `subscribeOn()` is more descriptive, specifying *how* a subscription should be handled concurrently. Reactor doesn't force any particular concurrency model; it's through `subscribeOn()` that you can specify the concurrency model, using one of the static methods from `Schedulers`, that you want to use. In this example, you used `parallel()`, which uses worker threads from a fixed pool (sized to the number of CPU cores). But `Schedulers` supports several concurrency models, such as those described in table 10.1.

Table 10.1 Concurrency models for Schedulers

Schedulers method	Description
<code>.immediate()</code>	Executes the subscription in the current thread.
<code>.single()</code>	Executes the subscription in a single, reusable thread. Reuses the same thread for all callers.

Table 10.1 Concurrency models for Schedulers (continued)

Schedulers method	Description
.newSingle()	Executes the subscription in a per-call dedicated thread.
.elastic()	Executes the subscription in a worker pulled from an unbounded, elastic pool. New worker threads are created as needed, and idle workers are disposed of (by default, after 60 seconds).
.parallel()	Executes the subscription in a worker pulled from a fixed-size pool, sized to the number of CPU cores.

The upside to using `flatMap()` and `subscribeOn()` is that you can increase the throughput of the stream by splitting the work across multiple parallel threads. But because the work is being done in parallel, with no guarantees on which will finish first, there's no way to know the order of items emitted in the resulting `Flux`. Therefore, `StepVerifier` is only able to verify that each item emitted exists in the expected list of `Player` objects and that there will be three such items before the `Flux` completes.

BUFFERING DATA ON A REACTIVE STREAM

In the course of processing the data flowing through a `Flux`, you might find it helpful to break the stream of data into bite-size chunks. The `buffer()` operation, shown in figure 10.18, can help with that.

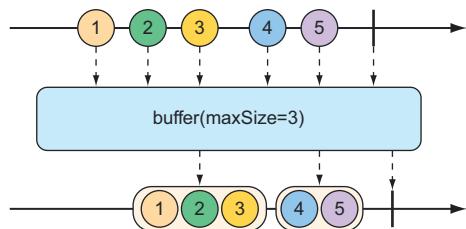


Figure 10.18 The `buffer` operation results in a `Flux` of lists of a given maximum size that are collected from the incoming `Flux`.

Given a `Flux` of `String` values, each containing the name of a fruit, you can create a new `Flux` of `List` collections, where each `List` has no more than a specified number of elements:

```

@Test
public void buffer() {
    Flux<String> fruitFlux = Flux.just(
        "apple", "orange", "banana", "kiwi", "strawberry");

    Flux<List<String>> bufferedFlux = fruitFlux.buffer(3);

    StepVerifier
        .create(bufferedFlux)
        .expectNext((Arrays.asList("apple", "orange", "banana")))
        .expectNext((Arrays.asList("kiwi", "strawberry")))
        .verifyComplete();
}

```

In this case, the Flux of String elements is buffered into a new Flux of List collections containing no more than three items each. Consequently, the original Flux that emits five String values will be converted to a Flux that emits two List collections, one containing three fruits and the other with two fruits.

So what? Buffering values from a reactive Flux into non-reactive List collections seems counterproductive. But when you combine `buffer()` with `flatMap()`, it enables each of the List collections to be processed in parallel:

```
Flux.just(
    "apple", "orange", "banana", "kiwi", "strawberry")
    .buffer(3)
    .flatMap(x ->
        Flux.fromIterable(x)
            .map(y -> y.toUpperCase())
            .subscribeOn(Schedulers.parallel())
            .log()
    ).subscribe();
```

In this new example, you still buffer a Flux of five String values into a new Flux of List collections. But then you apply `flatMap()` to that Flux of List collections. This takes each List buffer and creates a new Flux from its elements, and then applies a `map()` operation on it. Consequently, each buffered List is further processed in parallel in individual threads.

To prove that it works, I've also included a `log()` operation to be applied to each sub-Flux. The `log()` operation simply logs all Reactive Streams events, so that you can see what's really happening. As a result, the following entries are written to the log (with the time component removed for brevity's sake):

```
[main] INFO reactor.Flux.SubscribeOn.1 -
    onSubscribe(FluxSubscribeOn.SubscribeOnSubscriber)
[main] INFO reactor.Flux.SubscribeOn.1 - request(32)
[main] INFO reactor.Flux.SubscribeOn.2 -
    onSubscribe(FluxSubscribeOn.SubscribeOnSubscriber)
[main] INFO reactor.Flux.SubscribeOn.2 - request(32)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(APPLE)
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onNext(KIWI)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(ORANGE)
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onNext(STRAWBERRY)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(BANANA)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onComplete()
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onComplete()
```

As the log entries clearly show, the fruits in the first buffer (apple, orange, and banana) are handled in the `parallel-1` thread. Meanwhile, the fruits in the second buffer (kiwi and strawberry) are processed in the `parallel-2` thread. As is apparent by the fact that the log entries from each buffer are woven together, the two buffers are processed in parallel.

If, for some reason, you need to collect everything that a Flux emits into a List, you can call `buffer()` with no arguments:

```
Flux<List<String>> bufferedFlux = fruitFlux.buffer();
```

This results in a new Flux that emits a List that contains all the items published by the source Flux. You can achieve the same thing with the `collectList()` operation, illustrated by the marble diagram in figure 10.19.

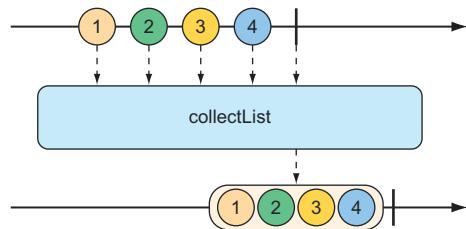


Figure 10.19 The collect-list operation results in a Mono containing a list of all messages emitted by the incoming Flux.

Rather than produce a Flux that publishes a List, `collectList()` produces a Mono that publishes a List. The following test method shows how it might be used:

```
@Test
public void collectList() {
    Flux<String> fruitFlux = Flux.just(
        "apple", "orange", "banana", "kiwi", "strawberry");

    Mono<List<String>> fruitListMono = fruitFlux.collectList();

    StepVerifier
        .create(fruitListMono)
        .expectNext(NSArray.asList(
            "apple", "orange", "banana", "kiwi", "strawberry"))
        .verifyComplete();
}
```

An even more interesting way of collecting items emitted by a Flux is to collect them into a Map. As shown in figure 10.20, the `collectMap()` operation results in a Mono that publishes a Map that's populated with entries whose key is calculated by a given Function.

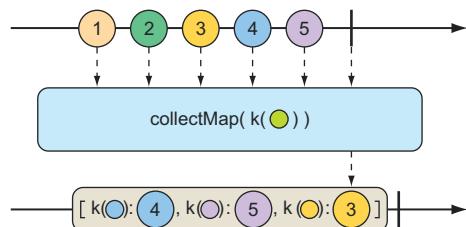


Figure 10.20 The collect-map operation results in a Mono containing a Map of messages emitted by the incoming Flux, where the key is derived from some characteristic of the incoming messages.

To see `collectMap()` in action, have a look at the following test method:

```
@Test
public void collectMap() {
    Flux<String> animalFlux = Flux.just(
        "aardvark", "elephant", "koala", "eagle", "kangaroo");

    Mono<Map<Character, String>> animalMapMono =
        animalFlux.collectMap(a -> a.charAt(0));

    StepVerifier
        .create(animalMapMono)
        .expectNextMatches(map -> {
            return
                map.size() == 3 &&
                map.get('a').equals("aardvark") &&
                map.get('e').equals("eagle") &&
                map.get('k').equals("kangaroo");
        })
        .verifyComplete();
}
```

The source Flux emits the names of a handful of animals. From that Flux, you use `collectMap()` to create a new Mono that emits a Map, where the key value is determined by the first letter of the animal name and the value is the animal name itself. In the event that two animal names start with the same letter (as with *elephant* and *eagle* or *koala* and *kangaroo*), the last entry flowing through the stream overrides any earlier entries.

10.3.4 Performing logic operations on reactive types

Sometimes you just need to know if the entries published by a Mono or Flux match some criteria. The `all()` and `any()` operations perform such logic. Figures 10.21 and 10.22 illustrate how `all()` and `any()` work.

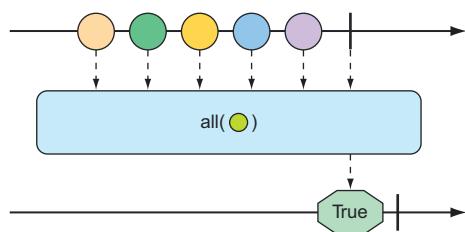


Figure 10.21 A flux can be tested to ensure that all messages meet some condition with the `all` operation.

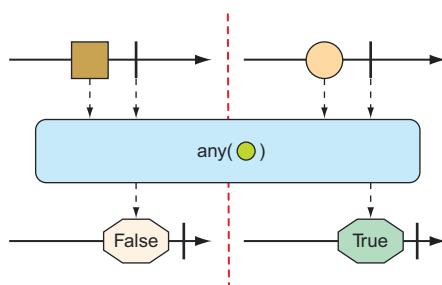


Figure 10.22 A flux can be tested to ensure that at least one message meets some condition with the `any` operation.

Suppose you want to know that every String published by a Flux contains the letter *a* or the letter *k*. The following test shows how to use all() to check for that condition:

```
@Test
public void all() {
    Flux<String> animalFlux = Flux.just(
        "aardvark", "elephant", "koala", "eagle", "kangaroo");

    Mono<Boolean> hasAMono = animalFlux.all(a -> a.contains("a"));
    StepVerifier.create(hasAMono)
        .expectNext(true)
        .verifyComplete();

    Mono<Boolean> hasKMono = animalFlux.all(a -> a.contains("k"));
    StepVerifier.create(hasKMono)
        .expectNext(false)
        .verifyComplete();
}
```

In the first StepVerifier, you check for the letter *a*. The all operation is applied to the source Flux, resulting in a Mono of type Boolean. In this case, all of the animal names contain the letter *a*, so true is emitted from the resulting Mono. But in the second StepVerifier, the resulting Mono will emit false because not all of the animal names contain a *k*.

Rather than perform an all-or-nothing check, maybe you're satisfied if at least one entry matches. In that case, the any() operation is what you want. This new test case uses any() to check for the letters *t* and *z*:

```
@Test
public void any() {
    Flux<String> animalFlux = Flux.just(
        "aardvark", "elephant", "koala", "eagle", "kangaroo");

    Mono<Boolean> hasAMono = animalFlux.any(a -> a.contains("t"));
    StepVerifier.create(hasAMono)
        .expectNext(true)
        .verifyComplete();

    Mono<Boolean> hasZMono = animalFlux.any(a -> a.contains("z"));
    StepVerifier.create(hasZMono)
        .expectNext(false)
        .verifyComplete();
}
```

In the first StepVerifier, you see that the resulting Mono emits true, because at least one animal name has the letter *t* (specifically, *elephant*). In the second case, the resulting Mono emits false, because none of the animal names contain *z*.

Summary

- Reactive programming involves creating pipelines through which data flows.
- The Reactive Streams specification defines four types: Publisher, Subscriber, Subscription, and Transformer (which is a combination of Publisher and Subscriber).
- Project Reactor implements Reactive Streams and abstracts stream definitions into two primary types, Flux and Mono, each of which offers several hundred operations.
- Spring 5 leverages Reactor to create reactive controllers, repositories, REST clients, and other reactive framework support.

11

Developing reactive APIs

This chapter covers

- Using Spring WebFlux
- Writing and testing reactive controllers and clients
- Consuming REST APIs
- Securing reactive web applications

Now that you've a good introduction to reactive programming and Project Reactor, you're ready to start applying those techniques in your Spring applications. In this chapter, we're going to revisit some of the controllers you wrote in chapter 6 to take advantage of Spring 5's reactive programming model.

More specifically, we're going to take a look at Spring 5's new reactive web framework—Spring WebFlux. As you'll quickly discover, Spring WebFlux is remarkably similar to Spring MVC, making it easy to apply, along with what you already know about building REST APIs in Spring.

11.1 Working with Spring WebFlux

Typical Servlet-based web frameworks, such as Spring MVC, are blocking and multi-threaded in nature, using a single thread per connection. As requests are handled,

a worker thread is pulled from a thread pool to process the request. Meanwhile, the request thread is blocked until it's notified by the worker thread that it's finished.

Consequently, blocking web frameworks won't scale effectively under heavy request volume. Latency in slow worker threads makes things even worse because it'll take longer for the worker thread to be returned to the pool, ready to handle another request. In some use cases, this arrangement is perfectly acceptable. In fact, this is largely how most web applications have been developed for well over a decade. But times are changing.

The clients of those web applications have grown from people occasionally viewing websites to people frequently consuming content and using applications that coordinate with HTTP APIs. And these days, the so-called *Internet of Things* (where humans aren't even involved) yields cars, jet engines, and other non-traditional clients constantly exchanging data with web APIs. With an increasing number of clients consuming web applications, scalability is more important than ever.

Asynchronous web frameworks, in contrast, achieve higher scalability with fewer threads—generally one per CPU core. By applying a technique known as *event looping* (as illustrated in figure 11.1), these frameworks are able to handle many requests per thread, making the per-connection cost more economical.

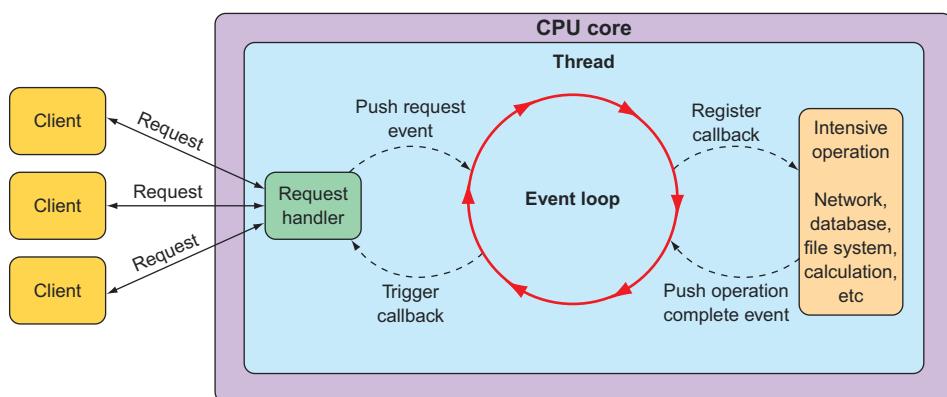


Figure 11.1 Asynchronous web frameworks apply event looping to handle more requests with fewer threads.

In an event loop, everything is handled as an event, including requests and callbacks from intensive operations like database and network operations. When a costly operation is needed, the event loop registers a callback for that operation to be performed in parallel, while it moves on to handle other events.

When the operation is complete, it's treated as an event by the event loop, the same as requests. As a result, asynchronous web frameworks are able to scale better under heavy request volume with fewer threads, resulting in reduced overhead for thread management.

Spring 5 has introduced a non-blocking, asynchronous web framework based largely on its Project Reactor to address the need for greater scalability in web applications and APIs. Let's take a look at Spring WebFlux—a reactive web framework for Spring.

11.1.1 Introducing Spring WebFlux

As the Spring team was considering how to add a reactive programming model to the web layer, it quickly became apparent that it would be difficult to do so without a great deal of work in Spring MVC. That would involve branching code to decide whether to handle requests reactively or not. In essence, the result would be two web frameworks packaged as one, with `if` statements to separate the reactive from the non-reactive.

Instead of trying to shoehorn a reactive programming model into Spring MVC, it was decided to create a separate reactive web framework, borrowing as much from Spring MVC as possible. Spring WebFlux is the result. Figure 11.2 illustrates the complete web development stack defined by Spring 5.

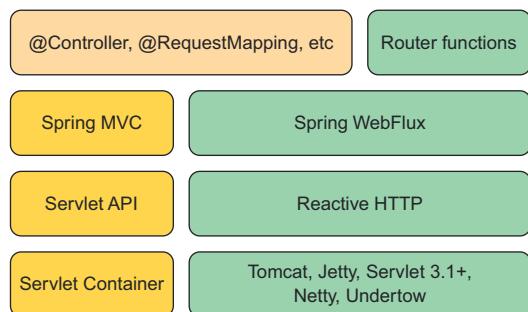


Figure 11.2 Spring 5 supports reactive web applications with a new web framework called **WebFlux**, which is a sibling to **Spring MVC** and shares many of its core components.

On the left side of figure 11.2, you see the Spring MVC stack that was introduced in version 2.5 of the Spring Framework. Spring MVC (covered in chapters 2 and 6) sits atop the Java Servlet API, which requires a servlet container (such as Tomcat) to execute on.

By contrast, Spring WebFlux (on the right side) doesn't have ties to the Servlet API, so it builds on top of a Reactive HTTP API, which is a reactive approximation of the same functionality provided by the Servlet API. And because Spring WebFlux isn't coupled to the Servlet API, it doesn't require a servlet container to run on. Instead, it can run on any non-blocking web container including Netty, Undertow, Tomcat, Jetty, or any Servlet 3.1 or higher container.

What's most noteworthy about figure 11.2 is the top left box, which represents the components that are common between Spring MVC and Spring WebFlux, primarily the annotations used to define controllers. Because Spring MVC and Spring WebFlux share the same annotations, Spring WebFlux is, in many ways, indistinguishable from Spring MVC.

The box in the top right corner represents an alternative programming model that defines controllers with a functional programming paradigm instead of using annotations. We'll talk more about Spring's functional web programming model in section 11.2.

The most significant difference between Spring MVC and Spring WebFlux boils down to which dependency you add to your build. When working with Spring WebFlux, you'll need to add the Spring Boot WebFlux starter dependency instead of the standard web starter (for example, `spring-boot-starter-web`). In the project's `pom.xml` file, it looks like this:

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

NOTE As with most of Spring Boot's starter dependencies, this starter can also be added to a project by checking the Reactive Web checkbox in the Initializr.

An interesting side-effect of using WebFlux instead of Spring MVC is that the default embedded server for WebFlux is Netty instead of Tomcat. Netty is one of a handful of asynchronous, event-driven servers and is a natural fit for a reactive web framework like Spring WebFlux.

Aside from using a different starter dependency, Spring WebFlux controller methods usually accept and return reactive types, like `Mono` and `Flux`, instead of domain types and collections. Spring WebFlux controllers can also deal with RxJava types like `Observable`, `Single`, and `Completable`.

REACTIVE SPRING MVC?

Although Spring WebFlux controllers typically return `Mono` and `Flux`, that doesn't mean that Spring MVC doesn't get to have some fun with reactive types. Spring MVC controller methods can also return a `Mono` or `Flux`, if you'd like.

The difference is in how those types are used. Whereas Spring WebFlux is a truly reactive web framework, allowing for requests to be handled in an event loop, Spring MVC is Servlet-based, relying on multithreading to handle multiple requests.

Let's put Spring WebFlux to work by rewriting some of Taco Cloud's API controllers to take advantage of Spring WebFlux.

11.1.2 Writing reactive controllers

You may recall that in chapter 6, you created a few controllers for Taco Cloud's REST API. Those controllers had request-handling methods that dealt with input and output in terms of domain types (such as `Order` and `Taco`) or collections of those domain types. As a reminder, consider the following snippet from `DesignTacoController` that you wrote back in chapter 6:

```
@RestController
@RequestMapping(path="/design",
    produces="application/json")
```

```
@CrossOrigin(origins="*")
public class DesignTacoController {

    ...

    @GetMapping("/recent")
    public Iterable<Taco> recentTacos() {
        PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
        return tacoRepo.findAll(page).getContent();
    }

    ...
}

}
```

As written, the `recentTacos()` controller handles HTTP GET requests for `/design/recent` to return a list of recently created tacos. More specifically, it returns an `Iterable` of type `Taco`. That's primarily because that's what's returned from the repository's `findAll()` method, or, more accurately, from the `getContent()` method on the `Page` object returned from `findAll()`.

That works fine, but `Iterable` isn't a reactive type. You won't be able to apply any reactive operations on it, nor can you let the framework take advantage of it as a reactive type to split any work over multiple threads. What you'd like is for `recentTacos()` to return a `Flux<Taco>`.

A simple, but somewhat limited option here is to rewrite `recentTacos()` to convert the `Iterable` to a `Flux`. And, while you're at it, you can do away with the paging code and replace it with a call to `take()` on the `Flux`:

```
@GetMapping("/recent")
public Flux<Taco> recentTacos() {
    return Flux.fromIterable(tacoRepo.findAll()).take(12);
}
```

Using `Flux.fromIterable()`, you convert the `Iterable<Taco>` to a `Flux<Taco>`. And now that you're working with a `Flux`, you can use the `take()` operation to limit the returned `Flux` to 12 `Taco` objects at most. Not only is the code simpler, it also deals with a reactive `Flux` rather than a plain `Iterable`.

Writing reactive code has been a winning move so far. But it would be even better if the repository gave you a `Flux` to start with so that you wouldn't need to do the conversion. If that were the case, then `recentTacos()` could be written to look like this:

```
@GetMapping("/recent")
public Flux<Taco> recentTacos() {
    return tacoRepo.findAll().take(12);
}
```

That's even better! Ideally, a reactive controller will be the tip of a stack that's reactive end to end, including controllers, repositories, the database, and any services that may sit in between. Such an end-to-end reactive stack is illustrated in figure 11.3.

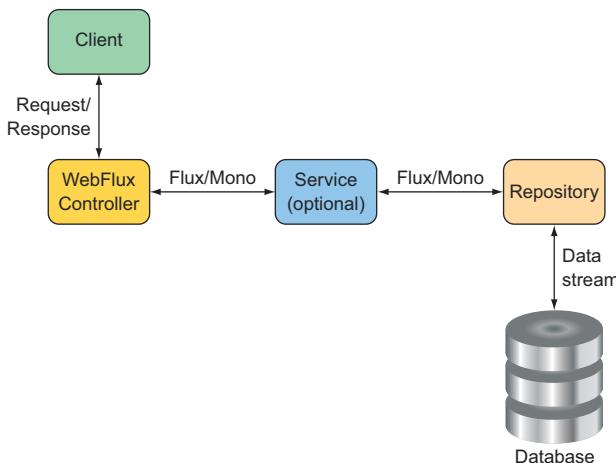


Figure 11.3 To maximize the benefit of a reactive web framework, it should be part of a full end-to-end reactive stack.

Such an end-to-end stack requires that the repository be written to return a `Flux` instead of an `Iterable`. We'll look into writing reactive repositories in the next chapter, but here's a sneak peek at what a reactive `TacoRepository` might look like:

```
public interface TacoRepository
    extends ReactiveCrudRepository<Taco, Long> { }
```

What's most important to note at this point, however, is that aside from working with a `Flux` instead of an `Iterable`, as well as how you obtain that `Flux`, the programming model for defining a reactive WebFlux controller is no different than for a non-reactive Spring MVC controller. Both are annotated with `@RestController` and a high-level `@RequestMapping` at the class level. And both have request-handling functions that are annotated with `@GetMapping` at the method level. It's truly a matter of what type the handler methods return.

Another important observation to make is that although you're getting a `Flux<Taco>` back from the repository, you can return it without calling `subscribe()`. Indeed, the framework will call `subscribe()` for you. This means that when a request for `/design/recent` is handled, the `recentTacos()` method will be called and will return before the data is even fetched from the database!

RETURNING SINGLE VALUES

As another example, consider the `tacoById()` method from the `DesignTacoController` as it was written in chapter 6:

```
@GetMapping("/{id}")
public Taco tacoById(@PathVariable("id") Long id) {
```

```

Optional<Taco> optTaco = tacoRepo.findById(id);
if (optTaco.isPresent()) {
    return optTaco.get();
}
return null;
}

```

Here, this method handles GET requests for `/design/{id}` and returns a single Taco object. Because the repository's `findById()` returns an `Optional`, you also had to write some clunky code to deal with that. But suppose for a minute that the `findById()` returns a `Mono<Taco>` instead of an `Optional<Taco>`. In that case, you can rewrite the controller's `tacoById()` to look like this:

```

@GetMapping("/{id}")
public Mono<Taco> tacoById(@PathVariable("id") Long id) {
    return tacoRepo.findById(id);
}

```

Wow! That's a lot simpler. What's more important, however, is that by returning a `Mono<Taco>` instead of a Taco, you're enabling Spring WebFlux to handle the response in a reactive manner. Consequently, your API will scale better in response to heavy loads.

WORKING WITH RXJAVA TYPES

It's worth pointing out that although Reactor types like `Flux` and `Mono` are a natural choice when working with Spring WebFlux, you can also choose to work with RxJava types like `Observable` and `Single`. For example, suppose that there's a service sitting between `DesignTacoController` and the backend repository that deals in terms of RxJava types. In that case, the `recentTacos()` method might be written like this:

```

@GetMapping("/recent")
public Observable<Taco> recentTacos() {
    return tacoService.getRecentTacos();
}

```

Similarly, the `tacoById()` method could be written to deal with an RxJava `Single` rather than a `Mono`:

```

@GetMapping("/{id}")
public Single<Taco> tacoById(@PathVariable("id") Long id) {
    return tacoService.lookupTaco(id);
}

```

In addition, Spring WebFlux controller methods can also return RxJava's `Completable`, which is equivalent to a `Mono<Void>` in Reactor. WebFlux can also return a `Flowable` as an alternative to `Observable` or Reactor's `Flux`.

HANDLING INPUT REACTIVELY

So far, we've only concerned ourselves with what reactive types the controller methods return. But with Spring WebFlux, you can also accept a `Mono` or a `Flux` as input to a

handler method. To demonstrate, consider the original implementation of `postTaco()` from `DesignTacoController`:

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
    return tacoRepo.save(taco);
}
```

As originally written, `postTaco()` not only returns a simple `Taco` object, but also accepts a `Taco` object that's bound to the content in the body of the request. This means that `postTaco()` can't be invoked until the request payload has been fully resolved and used to instantiate a `Taco` object. It also means `postTaco()` can't return until the blocking call to the repository's `save()` method returns. In short, the request is blocked twice: as it enters `postTaco()` and again, inside of `postTaco()`. But by applying a little reactive coding to `postTaco()`, you can make it a fully non-blocking, request-handling method:

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Mono<Taco> postTaco(@RequestBody Mono<Taco> tacoMono) {
    return tacoRepo.saveAll(tacoMono).next();
}
```

Here, `postTaco()` accepts a `Mono<Taco>` and calls the repository's `saveAll()` method, which, as you'll see in the next chapter, accepts any implementation of Reactive Streams' Publisher, including `Mono` or `Flux`. The `saveAll()` method returns a `Flux<Taco>`, but because you started with a `Mono`, you know there's at most one `Taco` that will be published by the `Flux`. You can therefore call `next()` to obtain a `Mono<Taco>` that will return from `postTaco()`.

By accepting a `Mono<Taco>` as input, the method is invoked immediately without waiting for the `Taco` to be resolved from the request body. And because the repository is also reactive, it'll accept a `Mono` and immediately return a `Flux<Taco>`, from which you call `next()` and return the resulting `Mono<Taco>` ... all before the request is even processed!

Spring WebFlux is a fantastic alternative to Spring MVC, offering the option of writing reactive web applications using the same development model as Spring MVC. But Spring 5 has another new trick up its sleeve. Let's take a look at how to create reactive APIs using Spring 5's new functional programming style.

11.2 Defining functional request handlers

Spring MVC's annotation-based programming model has been around since Spring 2.5 and is widely popular. It comes with a few downsides, however.

First, any annotation-based programming involves a split in the definition of *what* the annotation is supposed to do and *how* it's supposed to do it. Annotations themselves

define the what; the how is defined elsewhere in the framework code. This complicates the programming model when it comes to any sort of customization or extension because such changes require working in code that's external to the annotation. Moreover, debugging such code is tricky because you can't set a breakpoint on an annotation.

Also, as Spring continues to grow in popularity, developers new to Spring from other languages and frameworks may find annotation-based Spring MVC (and WebFlux) quite unlike what they already know. As an alternative to WebFlux, Spring 5 has introduced a new functional programming model for defining reactive APIs.

This new programming model is used more like a library and less like a framework, letting you map requests to handler code without annotations. Writing an API using Spring's functional programming model involves four primary types:

- RequestPredicate—Declares the kind(s) of requests that will be handled
- RouterFunction—Declares how a matching request should be routed to handler code
- ServerRequest—Represents an HTTP request, including access to header and body information
- ServerResponse—Represents an HTTP response, including header and body information

As a simple example that pulls all of these types together, consider the following Hello World example:

```
package demo;
import static org.springframework.web.reactive.function.server.RequestPredicates.GET;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;
import static org.springframework.web.reactive.function.server.ServerResponse.ok;
import static reactor.core.publisher.Mono.just;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RouterFunction;

@Configuration
public class RouterFunctionConfig {

    @Bean
    public RouterFunction<?> helloRouterFunction() {
        return route(GET("/hello"),
            request -> ok().body(just("Hello World!"), String.class));
    }
}
```

The first thing to notice is that you've chosen to statically import a few helper classes that you can use to create the aforementioned functional types. You've also statically imported Mono to keep the rest of the code easier to read and understand.

In this @Configuration class, you have a single @Bean method of type RouterFunction<?>. As mentioned, a RouterFunction declares mappings between one or more RequestPredicate objects and the functions that will handle the matching request(s).

The route() method from RouterFunctions accepts two parameters: a RequestPredicate and a function to handle matching requests. In this case, the GET() method from RequestPredicates declares a RequestPredicate that matches HTTP GET requests for the /hello path.

As for the handler function, it's written as a lambda, although it can also be a method reference. Although it isn't explicitly declared, the handler lambda accepts a ServerRequest as a parameter. It returns a ServerResponse using ok() from ServerResponse and body() from BodyBuilder, which was returned from ok(). This was done to create a response with an HTTP 200 (OK) status code and a body payload that says Hello World!

As written, the helloRouterFunction() method declares a RouterFunction that only handles a single kind of request. But if you need to handle a different kind of request, you don't have to write another @Bean method, although you can. You only need to call andRoute() to declare another RequestPredicate-to-function mapping. For example, here's how you might add another handler for GET requests for /bye:

```
@Bean
public RouterFunction<?> helloRouterFunction() {
    return route(GET("/hello"),
        request -> ok().body(just("Hello World!"), String.class))
    .andRoute(GET("/bye"),
        request -> ok().body(just("See ya!"), String.class));
}
```

Hello World samples are fine for dipping your toes into something new. But let's amp it up a bit and see how to use Spring's functional web programming model to handle requests that resemble real-world scenarios.

To demonstrate how the functional programming model might be used in a real-world application, let's reinvent the functionality of DesignTacoController in the functional style. The following configuration class is a functional analog to DesignTacoController:

```
@Configuration
public class RouterFunctionConfig {

    @Autowired
    private TacoRepository tacoRepo;

    @Bean
    public RouterFunction<?> routerFunction() {
        return route(GET("/design/taco"), this::recents)
```

```

        .andRoute(POST("/design"), this::postTaco);
    }

    public Mono<ServerResponse> recentTacos(ServerRequest request) {
        return ServerResponse.ok()
            .body(tacoRepo.findAll().take(12), Taco.class);
    }

    public Mono<ServerResponse> postTaco(ServerRequest request) {
        Mono<Taco> taco = request.bodyToMono(Taco.class);
        Mono<Taco> savedTaco = tacoRepo.save(taco);
        return ServerResponse
            .created(URI.create(
                "http://localhost:8080/design/taco/" +
                savedTaco.getId()))
            .body(savedTaco, Taco.class);
    }
}

```

As you can see, the `routerFunction()` method declares a `RouterFunction<?>` bean, like the Hello World example. But it differs in what types of requests are handled and how they're handled. In this case, the `RouterFunction` is created to handle GET requests for `/design/taco` and POST requests for `/design`.

What stands out even more is that the routes are handled by method references. Lambdas are great when the behavior behind a `RouterFunction` is relatively simple and brief. In many cases, however, it's better to extract that functionality into a separate method (or even into a separate method in a separate class) to maintain code readability.

For your needs, GET requests for `/design/taco` will be handled by the `recentTacos()` method. It uses the injected `TacoRepository` to fetch a `Mono<Taco>` from which it takes 12 items. And POST requests for `/design` are handled by the `postTaco()` method, which extracts a `Mono<Taco>` from the incoming `ServerRequest`. The `postTaco()` method then uses the `TacoRepository` to save it before responding with the `Mono<Taco>` that's returned from the `save()` method.

11.3 Testing reactive controllers

When it comes to testing reactive controllers, Spring 5 hasn't left us in the lurch. Indeed, Spring 5 has introduced `WebTestClient`, a new test utility that makes it easy to write tests for reactive controllers written with Spring WebFlux. To see how to write tests with `WebTestClient`, let's start by using it to test the `recentTacos()` method from `DesignTacoController` that you wrote in section 11.1.2.

11.3.1 Testing GET requests

One thing we'd like to assert about the `recentTacos()` method is that if an HTTP GET request is issued for the path `/design/recent`, then the response will contain a JSON payload with no more than 12 tacos. The test class in the next listing is a good start.

Listing 11.1 Using WebTestClient to test DesignTacoController

```

package tacos;

import static org.mockito.Mockito.*;
import java.util.ArrayList;
import java.util.List;
import org.junit.Test;
import org.mockito.Mockito;
import org.springframework.http.MediaType;
import org.springframework.test.web.reactive.server.WebTestClient;
import reactor.core.publisher.Flux;
import tacos.Ingredient.Type;
import tacos.data.TacoRepository;
import tacos.web.api.DesignTacoController;

```

```

public class DesignTacoControllerTest {

    @Test
    public void shouldReturnRecentTacos() {
        Taco[] tacos = {
            testTaco(1L), testTaco(2L),
            testTaco(3L), testTaco(4L),
            testTaco(5L), testTaco(6L),
            testTaco(7L), testTaco(8L),
            testTaco(9L), testTaco(10L),
            testTaco(11L), testTaco(12L),
            testTaco(13L), testTaco(14L),
            testTaco(15L), testTaco(16L)};
        Flux<Taco> tacoFlux = Flux.just(tacos);

        TacoRepository tacoRepo = Mockito.mock(TacoRepository.class);
        when(tacoRepo.findAll()).thenReturn(tacoFlux);
    }
}

WebTestClient testClient = WebTestClient.bindToController(
    new DesignTacoController(tacoRepo))
    .build();

testClient.get().uri("/design/recent")
    .exchange()
    .expectStatus().isOk()
    .expectBody()
        .jsonPath("$.id").isArray()
        .jsonPath("$.id").isNotEmpty()
        .jsonPath("$.id").isEqualTo(tacos[0].getId().toString())
        .jsonPath("$.name").isEqualTo("Taco 1").jsonPath("$.id")
        .isEqualTo(tacos[1].getId().toString()).jsonPath("$.id")
        .isEqualTo("Taco 2").jsonPath("$.id")
        .isEqualTo(tacos[11].getId().toString())

```

Creates some test data

Mocks TacoRepository

Creates a WebTestClient

Requests recent tacos

Verifies expected response

```

        .jsonPath("$.name").isEqualTo("Taco 12").jsonPath("$.name")
        .doesNotExist();
        .jsonPath("$.name").doesNotExist();
    }

    ...
}

}

```

The first thing that the `shouldReturnRecentTacos()` method does is set up test data in the form of a `Flux<Taco>`. This `Flux` is then provided as the return value from the `findAll()` method of a mock `TacoRepository`.

With regard to the `Taco` objects that will be published by `Flux`, they're created with a utility method named `testTaco()` that, when given a number, produces a `Taco` object whose ID and name are based on that number. The `testTaco()` method is implemented as follows:

```

private Taco testTaco(Long number) {
    Taco taco = new Taco();
    taco.setId(UUID.randomUUID());
    taco.setName("Taco " + number);
    List<IngredientUDT> ingredients = new ArrayList<>();
    ingredients.add(
        new IngredientUDT("INGA", "Ingredient A", Type.WRAP));
    ingredients.add(
        new IngredientUDT("INGB", "Ingredient B", Type.PROTEIN));
    taco.setIngredients(ingredients);
    return taco;
}

```

For the sake of simplicity, all test tacos will have the same two ingredients. But their ID and name will be determined by the given number.

Meanwhile, back in the `shouldReturnRecentTacos()` method, you instantiated a `DesignTacoController`, injecting the mock `TacoRepository` into the constructor. The controller is given to `WebTestClient.bindToController()` to create an instance of `WebTestClient`.

With all of the setup complete, you're now ready to use `WebTestClient` to submit a GET request to `/design/recent` and verify that the response meets your expectations. Calling `get().uri("/design/recent")` describes the request you want to issue. Then a call to `exchange()` submits the request, which will be handled by the controller that `WebTestClient` is bound to—the `DesignTacoController`.

Finally, you can affirm that the response is as expected. By calling `expectStatus()`, you assert that the response has an HTTP 200 (OK) status code. After that you see several calls to `jsonPath()` that assert that the JSON in the response body has the values it should have. The final assertion checks that the 12th element (in a zero-based array) is nonexistent, as the result should never have more than 12 elements.

If the JSON returns are complex, with a lot of data or highly nested data, it can be tedious to use `jsonPath()`. In fact, I left out many of the calls to `jsonPath()` in

listing 11.1 to conserve space. For those cases where it may be clumsy to use `jsonPath()`, `WebTestClient` offers `json()`, which accepts a `String` parameter containing the JSON to compare the response against.

For example, suppose that you've created the complete response JSON in a file named `recent-tacos.json` and placed it in the classpath under the path `/tacos`. Then you can rewrite the `WebTestClient` assertions to look like this:

```
ClassPathResource recentsResource =
    new ClassPathResource("/tacos/recent-tacos.json");
String recentsJson = StreamUtils.copyToString(
    recentsResource.getInputStream(), Charset.defaultCharset());

testClient.get().uri("/design/recent")
    .accept(MediaType.APPLICATION_JSON)
    .exchange()
    .expectStatus().isOk()
    .expectBody()
    .json(recentsJson);
```

Because `json()` accepts a `String`, you must first load the classpath resource into a `String`. Thankfully, Spring's `StreamUtils` makes this easy with `copyToString()`. The `String` that's returned from `copyToString()` will contain the entire JSON you expect in the response to your request. Giving it to the `json()` method ensures that the controller is producing the correct output.

Another option offered by `WebTestClient` allows you to compare the response body with a list of values. The `expectBodyList()` method accepts either a `Class` or a `ParameterizedTypeReference` indicating the type of elements in the list and returns a `ListBodySpec` object to make assertions against. Using `expectBodyList()`, you can rewrite the test to use a subset of the same test data you used to create the mock `TacoRepository`:

```
testClient.get().uri("/design/recent")
    .accept(MediaType.APPLICATION_JSON)
    .exchange()
    .expectStatus().isOk()
    .expectBodyList(Taco.class)
    .contains(Arrays.copyOf(tacos, 12));
```

Here you assert that the response body contains a list that has the same elements as the first 12 elements of the original `Taco` array you created at the beginning of the test method.

11.3.2 Testing POST requests

`WebTestClient` can do more than just test GET requests against controllers. It can also be used to test any kind of HTTP method, including GET, POST, PUT, PATCH, DELETE, and HEAD requests. Table 11.1 maps HTTP methods to `WebTestClient` methods.

Table 11.1 WebTestClient tests any kind of request against Spring WebFlux controllers.

HTTP Method	WebTestClient method
GET	.get()
POST	.post()
PUT	.put()
PATCH	.patch()
DELETE	.delete()
HEAD	.head()

As an example of testing another HTTP method request against a Spring WebFlux controller, let's look at another test against `DesignTacoController`. This time, you'll write a test of your API's taco creation endpoint by submitting a POST request to `/design`:

```
@Test
public void shouldSaveATaco() {
    TacoRepository tacoRepo = Mockito.mock(
        TacoRepository.class); ← Sets up test data
    Mono<Taco> unsavedTacoMono = Mono.just(testTaco(null));
    Taco savedTaco = testTaco(null);
    savedTaco.setId(1L);
    Mono<Taco> savedTacoMono = Mono.just(savedTaco);
    when(tacoRepo.save(any())).thenReturn(savedTacoMono); ← Mocks TacoRepository

    WebTestClient testClient = WebTestClient.bindToController(
        new DesignTacoController(tacoRepo)).build(); ← Creates WebTestClient

    testClient.post() ← POSTs a taco
        .uri("/design")
        .contentType(MediaType.APPLICATION_JSON)
        .body(unsavedTacoMono, Taco.class)
        .exchange()
        .expectStatus().isCreated() ← Verifies response
        .expectBody(Taco.class)
        .isEqualTo(savedTaco);
}
```

As with the previous test method, `shouldSaveATaco()` starts by setting up some test data, mocking `TacoRepository`, and building a `WebTestClient` that's bound to the controller. Then, it uses the `WebTestClient` to submit a POST request to `/design`, with a body of type `application/json` and a payload that's a JSON-serialized form of the `Taco` in the `unsaved Mono`. After performing `exchange()`, the test asserts that the response has an HTTP 201 (CREATED) status and a payload in the body equal to the saved `Taco` object.

11.3.3 Testing with a live server

The tests you've written so far have relied on a mock implementation of the Spring WebFlux framework so that a real server wouldn't be necessary. But you may need to test a WebFlux controller in the context of a server like Netty or Tomcat and maybe with a repository or other dependencies. That is to say, you may want to write an integration test.

To write a `WebTestClient` integration test, you start by annotating the test class with `@RunWith` and `@SpringBootTest` like any other Spring Boot integration test:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class DesignTacoControllerWebTest {

    @Autowired
    private WebTestClient testClient;

}
```

By setting the `webEnvironment` attribute to `WebEnvironment.RANDOM_PORT`, you're asking Spring to start a running server listening on a randomly chosen port.¹

You'll notice that you've also autowired a `WebTestClient` into the test class. This not only means that you'll no longer have to create one in your test methods, but also that you won't need to specify a full URL when making requests. That's because the `WebTestClient` will be rigged to know which port the test server is running on. Now you can rewrite `shouldReturnRecentTacos()` as an integration test that uses the autowired `WebTestClient`:

```
@Test
public void shouldReturnRecentTacos() throws IOException {
    testClient.get().uri("/design/recent")
        .accept(MediaType.APPLICATION_JSON).exchange()
        .expectStatus().isOk()
        .expectBody()
            .jsonPath("$.?[(@.id == 'TACO1')].name")
                .isEqualTo("Carnivore")
            .jsonPath("$.?[(@.id == 'TACO2')].name")
                .isEqualTo("Bovine Bounty")
            .jsonPath("$.?[(@.id == 'TACO3')].name")
                .isEqualTo("Veg-Out");
}
```

You've no doubt noticed that this new version of `shouldReturnRecentTacos()` has much less code. There's no longer any need to create a `WebTestClient` because you'll be making use of the autowired instance. And there's no need to mock `TacoRepository` because Spring will create an instance of `DesignTacoController` and inject it with a

¹ You could have also set `webEnvironment` to `WebEnvironment.DEFINED_PORT` and specified a port with the `properties` attribute, but that's generally inadvisable. Doing so opens the risk of a port clash with a concurrently running server.

real TacoRepository. In this new version of the test method, you use JSONPath expressions to verify values served from the database.

WebTestClient is useful when, in the course of a test, you need to consume the API exposed by a WebFlux controller. But what about when your application itself consumes some other API? Let's turn our attention to the client side of Spring's reactive web story and see how WebClient provides a REST client that deals in reactive types such as Mono and Flux.

11.4 Consuming REST APIs reactively

In chapter 7, you used RestTemplate to make client requests to the Taco Cloud API. RestTemplate is an old-timer, having been introduced in Spring version 3.0. In its time, it has been used to make countless requests on behalf of the applications that employ it.

But all of the methods provided by RestTemplate deal in non-reactive domain types and collections. This means that if you want to work with a response's data in a reactive way, you'll need to wrap it with a Flux or Mono. And if you already have a Flux or Mono and you want to send it in a POST or PUT request, then you'll need to extract the data into a non-reactive type before making the request.

It would be nice if there was a way to use RestTemplate natively with reactive types. Fear not. Spring 5 offers WebClient as a reactive alternative to RestTemplate. WebClient lets you both send and receive reactive types when making requests to external APIs.

Using WebClient is quite different from using RestTemplate. Rather than have several methods to handle different kinds of requests, WebClient has a fluent builder-style interface that lets you describe and send requests. The general usage pattern for working with WebClient is

- Create an instance of WebClient (or inject a WebClient bean)
- Specify the HTTP method of the request to send
- Specify the URI and any headers that should be in the request
- Submit the request
- Consume the response

Let's look at several examples of WebClient in action, starting with how to use WebClient to send HTTP GET requests.

11.4.1 GETting resources

As an example of WebClient usage, suppose that you need to fetch an Ingredient object by its ID from the Taco Cloud API. Using RestTemplate, you might use the getForObject() method. But with WebClient, you build the request, retrieve a response, and then extract a Mono that publishes the Ingredient object:

```
Mono<Ingredient> ingredient = WebClient.create()
    .get()
```

```

.uri("http://localhost:8080/ingredients/{id}", ingredientId)
.retrieve()
.bodyToMono(Ingredient.class);

ingredient.subscribe(i -> { ... })

```

Here you create a new `WebClient` instance with `create()`. Then you use `get()` and `uri()` to define a GET request to `http://localhost:8080/ingredients/{id}`, where the `{id}` placeholder will be replaced by the value in `ingredientId`. The `retrieve()` method executes the request. Finally, a call to `bodyToMono()` extracts the response's body payload into a `Mono<Ingredient>` on which you can continue applying additional `Mono` operations.

To apply additional operations on the `Mono` returned from `bodyToMono()`, it's important to subscribe to it before the request will even be sent. Making requests that can return a collection of values is as easy. For example, the following snippet of code fetches all ingredients:

```

Flux<Ingredient> ingredients = WebClient.create()
    .get()
    .uri("http://localhost:8080/ingredients")
    .retrieve()
    .bodyToFlux(Ingredient.class);

ingredients.subscribe(i -> { ... })

```

For the most part, fetching multiple items is the same as making a request for a single item. The big difference is that instead of using `bodyToMono()` to extract the response's body into a `Mono`, you use `bodyToFlux()` to extract it into a `Flux`.

As with `bodyToMono()`, the `Flux` returned from `bodyToFlux()` hasn't yet been subscribed to. This allows additional operations (filters, maps, and so forth) to be applied to the `Flux` before the data starts flowing through it. Therefore, it's important to subscribe to the resulting `Flux` or else the request will never even be sent.

MAKING REQUESTS WITH A BASE URI

You may find yourself using a common base URI for many different requests. In that case, it can be useful to create a `WebClient` bean with a base URI and inject it anywhere it's needed. Such a bean could be declared like this:

```

@Bean
public WebClient webClient() {
    return WebClient.create("http://localhost:8080");
}

```

Then, anywhere you need to make requests using that base URI, the `WebClient` bean can be injected and used like this:

```

@.Autowired
WebClient webClient;

```

```
public Mono<Ingredient> getIngredientById(String ingredientId) {
    Mono<Ingredient> ingredient = webClient
        .get()
        .uri("/ingredients/{id}", ingredientId)
        .retrieve()
        .bodyToMono(Ingredient.class);

    ingredient.subscribe(i -> { ... })
}
```

Because the `WebClient` had already been created, you're able to get right to work by calling `get()`. As for the URI, you need to specify only the path relative to the base URI when calling `uri()`.

TIMING OUT ON LONG-RUNNING REQUESTS

One thing that you can count on is that networks aren't always reliable or as fast as you'd expect them to be. Or maybe a remote server is sluggish in handling a request. Ideally, a request to a remote service will return in a reasonable amount of time. But if not, it would be great if the client didn't get stuck waiting on a response for too long.

To avoid having your client requests held up by a sluggish network or service, you can use the `timeout()` method from `Flux` or `Mono` to put a limit on how long you'll wait for data to be published. As an example, consider how you might use `timeout()` when fetching ingredient data:

```
Flux<Ingredient> ingredients = WebClient.create()
    .get()
    .uri("http://localhost:8080/ingredients")
    .retrieve()
    .bodyToFlux(Ingredient.class);

ingredients
    .timeout(Duration.ofSeconds(1))
    .subscribe(
        i -> { ... },
        e -> {
            // handle timeout error
        })
}
```

As you can see, before subscribing to the `Flux`, you called `timeout()`, specifying a duration of 1 s. If the request can be fulfilled in less than 1 s, then there's no problem. But if the request is taking longer than 1 s, it'll timeout and the error handler given as the second parameter to `subscribe()` is invoked.

11.4.2 Sending resources

Sending data with `WebClient` isn't much different from receiving data. As an example, let's say that you've a `Mono<Ingredient>` and want to send a POST request with the `Ingredient` that's published by the `Mono` to the URI with a relative path of `/ingredients`.

All you must do is use the `post()` method instead of `get()` and specify that the `Mono` is to be used to populate the request body by calling `body()`:

```
Mono<Ingredient> ingredientMono = ...;

Mono<Ingredient> result = webClient
    .post()
    .uri("/ingredients")
    .body(ingredientMono, Ingredient.class)
    .retrieve()
    .bodyToMono(Ingredient.class);

result.subscribe(i -> { ... })
```

If you don't have a `Mono` or `Flux` to send, but instead have the raw domain object on hand, you can use `syncBody()`. For example, suppose that instead of a `Mono<Ingredient>`, you have an `Ingredient` that you want to send in the request body:

```
Ingredient ingredient = ...;

Mono<Ingredient> result = webClient
    .post()
    .uri("/ingredients")
    .syncBody(ingredient)
    .retrieve()
    .bodyToMono(Ingredient.class);

result.subscribe(i -> { ... })
```

If instead of a POST request you want to update an `Ingredient` with a PUT request, you call `put()` instead of `post()` and adjust the URI path accordingly:

```
Mono<Void> result = webClient
    .put()
    .uri("/ingredients/{id}", ingredient.getId())
    .syncBody(ingredient)
    .retrieve()
    .bodyToMono(Void.class)
    .subscribe();
```

PUT requests typically have empty response payloads, so you must ask `bodyToMono()` to return a `Mono` of type `Void`. On subscribing to that `Mono`, the request will be sent.

11.4.3 Deleting resources

`WebClient` also allows the removal of resources by way of its `delete()` method. For example, the following code deletes an ingredient for a given ID:

```
Mono<Void> result = webClient
    .delete()
    .uri("/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Void.class)
    .subscribe();
```

As with PUT requests, DELETE requests don't typically have a payload. Once again, you return and subscribe to a `Mono<Void>` to send the request.

11.4.4 Handling errors

All of the `WebClient` examples thus far have assumed a happy ending; there were no responses with 400-level or 500-level status codes. Should either kind of error statuses be returned, `WebClient` will log the failure; otherwise, it'll silently ignore it.

If you need to handle such errors, then a call to `onStatus()` can be used to specify how various HTTP status codes should be dealt with. `onStatus()` accepts two functions: a predicate function, which is used to match the HTTP status, and a function that, given a `ClientResponse` object, returns a `Mono<Throwable>`.

To demonstrate how `onStatus()` can be used to create a custom error handler, consider the following use of `WebClient` that aims to fetch an ingredient given its ID:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);
```

As long as the value in `ingredientId` matches a known ingredient resource, then the resulting `Mono` will publish the `Ingredient` object when it's subscribed to. But what would happen if there were no matching ingredient?

When subscribing to a `Mono` or `Flux` that might end in error, it's important to register an error consumer as well as a data consumer in the call to `subscribe()`:

```
ingredientMono.subscribe(
    ingredient -> {
        // handle the ingredient data
        ...
    },
    error -> {
        // deal with the error
        ...
});
```

If the ingredient resource is found, then the first lambda (the data consumer) given to `subscribe()` is invoked with the matching `Ingredient` object. But if it isn't found, then the request responds with a status code of HTTP 404 (NOT FOUND), which results in the second lambda (the error consumer) being given by default a `WebClientResponseException`.

The biggest problem with `WebClientResponseException` is that it's rather non-specific as to what may have gone wrong to cause the `Mono` to fail. Its name suggests that there was an error in the response from a request made by `WebClient`, but you'll need to dig into `WebClientResponseException` to know what went wrong. And in any event, it would be nice if the exception given to the error consumer were more domain-specific instead of `WebClient`-specific.

By adding a custom error handler, you can provide code that translates a status code to a `Throwable` of your own choosing. Let's say that you want a failed request for an ingredient resource to result in the `Mono` completing in error with a `UnknownIngredientException`. You can add a call to `onStatus()` after the call to `retrieve()` to achieve that:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .onStatus(HttpStatus::is4xxClientError,
        response -> Mono.just(new UnknownIngredientException()))
    .bodyToMono(Ingredient.class);
```

The first argument in the `onStatus()` call is a predicate that's given an `HttpStatus` and returns `true` if the status code is one you want to handle. And if the status code matches, then the response will be returned to the function in the second argument to handle as it sees fit, ultimately returning a `Mono` of type `Throwable`.

In the example, if the status code is a 400-level status code (for example, a client error), then a `Mono` will be returned with an `UnknownIngredientException`. This causes the `ingredientMono` to fail with that exception.

Note that `HttpStatus::is4xxClientError` is a method reference to the `is4xxClientError` method of `HttpStatus`. It's this method that will be invoked on the given `HttpStatus` object. If you want, you can use another method on `HttpStatus` as a method reference; or you can provide your own function in the form of a lambda or method reference that returns a boolean.

For example, you can get even more precise in your error handling by checking specifically for an HTTP 404 (NOT FOUND) status by changing the call to `onStatus()` to look like this:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .onStatus(status -> status == HttpStatus.NOT_FOUND,
        response -> Mono.just(new UnknownIngredientException()))
    .bodyToMono(Ingredient.class);
```

It's also worth noting that you can have as many calls to `onStatus()` as you need to handle any variety of HTTP status codes that might come back in the response.

11.4.5 Exchanging requests

Up to this point, you've used the `retrieve()` method to signify sending a request when working with `WebClient`. In those cases, the `retrieve()` method returned an object of type `ResponseSpec`, through which you were able to handle the response with calls to methods such as `onStatus()`, `bodyToFlux()`, and `bodyToMono()`. Working with `ResponseSpec` is fine for simple cases, but it's limited in a few ways. If you need

access to the response's headers or cookie values, for example, then `ResponseSpec` isn't going to work for you.

When `ResponseSpec` comes up short, you can try calling `exchange()` instead of `retrieve()`. The `exchange()` method returns a `Mono` of type `ClientResponse`, on which you can apply reactive operations to inspect and use data from the entire response, including the payload, headers, and cookies.

Before we look at what makes `exchange()` different from `retrieve()`, let's start by looking at how similar they are. The following snippet of code uses a `WebClient` and `exchange()` to fetch a single ingredient by the ingredient's ID:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .exchange()
    .flatMap(cr -> cr.bodyToMono(Ingredient.class));
```

This is roughly equivalent to the following example that uses `retrieve()`:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);
```

In the `exchange()` example, rather than use the `ResponseSpec` object's `bodyToMono()` to get a `Mono<Ingredient>`, you get a `Mono<ClientResponse>` on which you can apply a flat-mapping function to map the `ClientResponse` to a `Mono<Ingredient>`, which is flattened into the resulting `Mono`.

Now let's see what makes `exchange()` different. Let's suppose that the response from the request might include a header named `X_UNAVAILABLE` with a value of `true` to indicate that (for some reason) the ingredient in question is unavailable. And for the sake of discussion, suppose that if that header exists, you want the resulting `Mono` to be empty—to not return anything. You can achieve this scenario by adding another call to `flatMap()` such that the entire `WebClient` call looks like this:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .exchange()
    .flatMap(cr -> {
        if (cr.headers().header("X_UNAVAILABLE").contains("true")) {
            return Mono.empty();
        }
        return Mono.just(cr);
    })
    .flatMap(cr -> cr.bodyToMono(Ingredient.class));
```

The new `flatMap()` call inspects the given `ClientRequest` object's headers, looking for a header named `X_UNAVAILABLE` with a value of `true`. If found, it returns an

empty Mono. Otherwise, it returns a new Mono that contains the ClientResponse. In either event, the Mono returned will be flattened into the Mono that the next flatMap() call will operate on.

11.5 Securing reactive web APIs

For as long as there has been Spring Security (and even before that when it was known as Acegi Security), its web security model has been built around servlet filters. After all, it just makes sense. If you need to intercept a request bound for a servlet-based web framework to ensure that the requester has proper authority, a servlet filter is an obvious choice. But Spring WebFlux puts a kink into that approach.

When writing a web application with Spring WebFlux, there's no guarantee that servlets are even involved. In fact, a reactive web application is debatably more likely to be built on Netty or some other non-servlet server. Does this mean that the servlet filter-based Spring Security can't be used to secure Spring WebFlux applications?

It's true that using servlet filters isn't an option when securing a Spring WebFlux application. But Spring Security is still up to the task. Starting with version 5.0.0, Spring Security can be used to secure both servlet-based Spring MVC and reactive Spring WebFlux applications. It does this using Spring's WebFilter, a Spring-specific analog to servlet filters that doesn't demand dependence on the servlet API.

What's even more remarkable, though, is that the configuration model for reactive Spring Security isn't much different from what you saw in chapter 4. In fact, unlike Spring WebFlux, which has a separate dependency from Spring MVC, Spring Security comes as the same Spring Boot security starter, regardless of whether you intend to use it to secure a Spring MVC web application or one written with Spring WebFlux. As a reminder, here's what the security starter looks like:

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

That said, there are a few small differences between Spring Security's reactive and non-reactive configuration models. It's worth taking a quick look at how the two configuration models compare.

11.5.1 Configuring reactive web security

As a reminder, configuring Spring Security to secure a Spring MVC web application typically involves creating a new configuration class that extends WebSecurityConfigurerAdapter and is annotated with @EnableWebSecurity. Such a configuration class would override a configuration() method to specify web security specifics such as what authorizations are required for certain request paths. The following simple Spring Security configuration class serves as a reminder of how to configure security for a non-reactive Spring MVC application:

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

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/design", "/orders").hasAuthority("USER")
            .antMatchers("/**").permitAll();
    }

}
```

Now let's see what this same configuration might look like for a reactive Spring WebFlux application. The following listing shows a reactive security configuration class that's roughly equivalent to the simple security configuration from before.

Listing 11.2 Configuring Spring Security for a Spring WebFlux application

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(
        ServerHttpSecurity http) {
        return http
            .authorizeExchange()
            .pathMatchers("/design", "/orders").hasAuthority("USER")
            .anyExchange().permitAll()
            .and()
            .build();
    }
}
```

As you can see, there's a lot that's familiar, while at the same time different. Rather than `@EnableWebSecurity`, this new configuration class is annotated with `@EnableWebFluxSecurity`. What's more, the configuration class doesn't extend `WebSecurityConfigurerAdapter` or any other base class whatsoever. Therefore, it also doesn't override any `configure()` methods.

In place of a `configure()` method, you declare a bean of type `SecurityWebFilterChain` with the `securityWebFilterChain()` method. The body of `securityWebFilterChain()` isn't much different from the previous configuration's `configure()` method, but there are some subtle changes.

Primarily, the configuration is declared using a given `ServerHttpSecurity` object instead of a `HttpSecurity` object. Using the given `ServerHttpSecurity`, you can call `authorizeExchange()`, which is roughly equivalent to `authorizeRequests()`, to declare request-level security.

NOTE `ServerHttpSecurity` is new to Spring Security 5 and is the reactive analog to `HttpSecurity`.

When matching paths, you can still use Ant-style wildcard paths, but do so with the `pathMatchers()` method instead of `antMatchers()`. And as a convenience, you no longer need to specify a catch-all Ant-style path of `/**` because the `anyExchange()` returns the catch-all you need.

Finally, because you're declaring the `SecurityWebFilterChain` as a bean instead of overriding a framework method, you must call the `build()` method to assemble all of the security rules into the `SecurityWebFilterChain` to be returned.

Aside from those small differences, configuring web security isn't that different for Spring WebFlux than for Spring MVC. But what about user details?

11.5.2 Configuring a reactive user details service

When extending `WebSecurityConfigurerAdapter`, you override one `configure()` method to declare web security rules and another `configure()` method to configure authentication logic, typically by defining a `UserDetails` object. As a reminder of what this looks like, consider the following overridden `configure()` method that uses an injected `UserRepository` object in an anonymous implementation of `UserDetailsService` to look up a user by username:

```
@Autowired
UserRepository userRepo;

@Override
protected void
    configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth
        .userDetailsService(new UserDetailsService() {
            @Override
            public UserDetails loadUserByUsername(String username)
                throws UsernameNotFoundException {
                User user = userRepo.findByUsername(username)
                    if (user == null) {
                        throw new UsernameNotFoundException(
                            "User " + not found")
                    }
                    return user.toUserDetails();
                }
            });
}
```

In this non-reactive configuration, you override the only method required by `UserDetailsService`, `loadUserByUsername()`. Inside of that method, you use the given `UserRepository` to look up the user by the given username. If the name isn't found, you throw a `UsernameNotFoundException`. But if it's found, then you call a helper method `toUserDetails()` to return the resulting `UserDetails` object.

In a reactive security configuration, you don't override a `configure()` method. Instead, you declare a `ReactiveUserDetailsService` bean. `ReactiveUserDetailsService` is the reactive equivalent to `UserDetailsService`. Like `UserDetailsService`, `ReactiveUserDetailsService` requires implementation of only a single method. Specifically, the `findByUsername()` method returns a `Mono<userDetails>` instead of a raw `UserDetails` object.

In the following example, the `ReactiveUserDetailsService` bean is declared to use a given `UserRepository`, which is presumed to be a reactive Spring Data repository (which we'll talk more about in the next chapter):

```
@Service
public ReactiveUserDetailsService userDetailsService(
    UserRepository userRepo) {
    return new ReactiveUserDetailsService() {
        @Override
        public Mono<UserDetails> findByUsername(String username) {
            return userRepo.findByUsername(username)
                .map(user -> {
                    return user.toUserDetails();
                });
        }
    };
}
```

Here, a `Mono<UserDetails>` is returned as required, but the `UserRepository.findByUsername()` method returns a `Mono<User>`. Because it's a `Mono`, you can chain operations on it, such as a `map()` operation to map the `Mono<User>` to a `Mono<UserDetails>`.

In this case, the `map()` operation is applied with a lambda that calls the helper `toUserDetails()` method on the `User` object published by the `Mono`. This converts the `User` to a `UserDetails`. As a consequence, the `.map()` operation returns a `Mono<UserDetails>`, which is precisely what the `ReactiveUserDetailsService.findByUsername()` requires.

Summary

- Spring WebFlux offers a reactive web framework whose programming model mirrors that of Spring MVC, even sharing many of the same annotations.
- Spring 5 also offers a functional programming model as an alternative to Spring WebFlux.
- Reactive controllers can be tested with `WebTestClient`.
- On the client-side, Spring 5 offers `WebClient`, a reactive analog to Spring's `RestTemplate`.
- Although WebFlux has some significant implications for the underlying mechanisms for securing a web application, Spring Security 5 supports reactive security with a programming model that isn't dramatically different from non-reactive Spring MVC applications.

Persisting data reactively

12

This chapter covers

- Spring Data's reactive repositories
- Writing reactive repositories for Cassandra and MongoDB
- Adapting non-reactive repositories for reactive use
- Data modeling with Cassandra

As I think about non-blocking reactive code and blocking imperative code, I start to think about rush hour. *Rush hour* is strangely named. Everybody seems to be in a rush to get where they're going, but usually they're all sitting near-motionless in traffic. If it weren't for everyone else on the road, I'd have no trouble getting to my destination.

Even though I'm eager to get somewhere (I'm non-blocking), that doesn't mean that someone else on the road isn't blocking me somehow. There may be other motorists ahead who have had a fender bender and are literally blocking the road for other commuters. So even though my efforts to get home are essentially non-blocking, I'm still blocked until the accident scene is cleared up.

In the previous chapter, you saw how to create reactive, non-blocking controllers with Spring WebFlux. This helps to improve scalability in the web layer. But those

controllers are only truly non-blocking if other components that they work with are also non-blocking. If we write Spring WebFlux controllers that still depend on blocking repositories, our reactive controllers will be blocked waiting for them to produce data.

Therefore, it's important that the entire flow of data, all the way from the controllers to the database, be reactive and non-blocking. In this chapter, you'll see how to write reactive repositories using Spring Data that follow a similar programming model as those you created in chapter 3. We'll start by taking a high-level survey of Spring Data's reactive support.

12.1 Understanding Spring Data's reactive story

Beginning with the Spring Data Kay release train, Spring Data offered its first support for working with reactive repositories. This includes support for a reactive programming model when persisting data with Cassandra, MongoDB, Couchbase, or Redis.

What's in a name?

Although Spring Data projects are versioned at their own pace, they're collectively published in a *release train*, where each version of the release train is named for a significant figure in computer science.

These names are alphabetical in nature and include names such as Babbage, Codd, Dijkstra, Evans, Fowler, Gosling, Hopper, and Ingalls. At the time this is being written, the most recent release train version is Spring Data Kay, named after Alan Kay, one of the designers of the Smalltalk programming language.

You may have noticed that I failed to mention relational databases or JPA. Unfortunately, there's no support for reactive JPA. Although relational databases are certainly the most prolific databases in the industry, supporting a reactive programming model with Spring Data JPA would require that the databases and JDBC drivers involved also support non-blocking reactive models. It's unfortunate that, at least for now, there's no support for working with relational databases reactively. Hopefully, this situation will be resolved in the near future.

In the meantime, this chapter focuses on using Spring Data to develop repositories that deal in reactive types for those databases that do support a reactive model. Let's see how Spring Data's reactive model compares to its non-reactive model.

12.1.1 Spring Data reactive distilled

The essence of Spring Data's reactive story can be summed up by saying that reactive repositories have methods that accept and return `Mono` and `Flux` instead of domain entities and collections. A repository method that fetches `Ingredient` objects by ingredient type from the backing database might be declared as follows in the repository interface:

```
Flux<Ingredient> findByType(Ingredient.Type type);
```

As you can see, this `findByType()` method returns a `Flux<Ingredient>` instead of a `List<Ingredient>` or an `Iterable<Ingredient>` as its non-reactive analog would.

Likewise, when saving a Taco, the repository would have a `saveAll()` method with the following signature:

```
<Taco> Flux<Taco> saveAll(Publisher<Taco> tacoPublisher);
```

In this case, the `saveAll()` method accepts a Publisher of type Taco (either a `Mono<Taco>` or a `Flux<Taco>`) and returns a `Flux<Taco>`. This is in contrast to a non-reactive repository, which would have a `save()` method that deals with the domain type directly, accepting a Taco object and returning the saved Taco object.

Put simply, Spring Data's reactive repositories share a near-identical programming model with Spring Data's non-reactive repositories, like those in chapter 3. The only material difference is that reactive repositories have methods that take and return `Flux` and `Mono` instead of raw domain types and collections.

12.1.2 Converting between reactive and non-reactive types

Before we look any further at how to write reactive repositories with Spring Data, let's take a moment to address the elephant in the room. You may have an existing relational database and it may not be practical to migrate your data to one of the four databases that Spring Data supports with its reactive programming model. Does that mean you can't apply reactive programming in your application at all?

Although the full benefit of reactive programming comes when you have a reactive model from end to end, including at the database level, there's still some benefit to be had by using reactive flows on top of a non-reactive database. Even though your chosen database doesn't support non-blocking reactive queries, you can still fetch data in a blocking fashion and then translate it into a reactive type as soon as possible for the benefit of upstream components.

Suppose, for example, that you're working with a relational database and using Spring Data JPA for persistence. Your `OrderRepository` may have a method with the following signature:

```
List<Order> findByUser(User user);
```

This method will return a non-reactive `List<Order>` containing all of the `Order` entities for a given `User`. When `findByUser()` is called, it will block while the query is executed and the results are collected into `List`. Because `List` isn't a reactive type, you won't be able to perform any of the operations afforded by `Flux` on it. Moreover, if the caller is a controller, it won't be able to work with the results reactively to achieve improved scalability.

You can't do anything about the blocking nature of invoking a method on a JPA repository. What you can do, however, is convert the non-reactive `List` into a `Flux` as

soon as you receive it, so that you can deal with the results reactively from there on. To do so, you simply use `Flux.fromIterable()`:

```
List<Order> orders = repo.findByUser(someUser);  
Flux<Order> orderFlux = Flux.fromIterable(orders);
```

Likewise, if you were to fetch a single `Order` by its ID, you could immediately convert it to a `Mono`:

```
Order order = repo.findById(Long id);  
Mono<Order> orderMono = Mono.just(order);
```

By using `Mono.just()` and the `fromIterable()`, `fromArray()`, and `fromStream()` methods of `Flux`, you can isolate the non-reactive blocking code in your repositories and deal with reactive types elsewhere in your application.

What about going the other way? What if you have a `Mono` or `Flux` given to you and you need to call `save()` on a non-reactive JPA repository? Fortunately, `Mono` and `Flux` both have operations to extract the data that they publish into domain types or an `Iterable`.

For example, suppose a `WebFlux` controller accepts a `Mono<Taco>`, and you need to save it using the `save()` method in a Spring Data JPA repository. No problem—just call the `block()` method on the `Mono` to extract the `Taco` object:

```
Taco taco = tacoMono.block();  
tacoRepo.save(taco);
```

As its name implies, the `block()` method will perform a blocking operation to perform the extractions.

As for extracting data from a `Flux`, you'll likely want to use `toIterable()`. Let's say you're given a `Flux<Taco>` and need to call `saveAll()` on a Spring Data JPA repository. The following snippet of code shows how to extract an `Iterable<Taco>` from a `Flux<Taco>` to do precisely that:

```
Iterable<Taco> tacos = tacoFlux.toIterable();  
tacoRepo.saveAll(tacos);
```

As with `Mono.block()`, `Flux.toIterable()` blocks as it collects all the objects published by the `Flux` into an `Iterable`. Because of their blocking nature, `Mono.block()` and `Flux.toIterable()` should be used sparingly and with the clear understanding that using them breaks out of the reactive programming model.

Another more reactive approach that avoids a blocking extraction operation is to subscribe to the `Mono` or `Flux` and perform the desired operation on each element as it's published. For example, to save all `Taco` objects published by a `Flux<Taco>` when the repository is non-reactive, you might do something like this:

```
tacoFlux.subscribe(taco -> {  
    tacoRepo.save(taco);  
});
```

Even though the call to the repository’s `save()` method is still a non-reactive blocking operation, using `subscribe()` is a more natural, reactive approach to consuming and processing the data published by a `Flux` or `Mono`.

But that’s enough talk about how to work with non-reactive repositories. Let’s start using the real power of Spring Data’s reactive support to create reactive repositories for the Taco Cloud application.

12.1.3 Developing reactive repositories

As you saw in chapter 3, one of the most amazing features of Spring Data is the ability to declare repository interfaces and have Spring Data automatically implement them at runtime. In that chapter, we focused primarily on Spring Data JPA, but the same programming model is applicable for nonrelational databases, including Cassandra and MongoDB.

Built on top of their non-reactive repository support, Spring Data Cassandra and Spring Data MongoDB both support a reactive model. With these databases in the backend providing data persistence, Spring applications can truly offer end-to-end reactive flows that span from the web layer to the database. Let’s start by looking at how to persist data to Cassandra using reactive Spring Data repositories.

12.2 Working with reactive Cassandra repositories

Cassandra is a distributed, high-performance, always available, eventually consistent, partitioned-row-store, NoSQL database.

That’s a mouthful of adjectives to describe a database, but each one accurately speaks to the power of working with Cassandra. To put it in simpler terms, Cassandra deals in rows of data, which are written to tables, which are partitioned across one-to-many distributed nodes. No single node carries all the data, but any given row may be replicated across multiple nodes, thus eliminating any single point of failure.

Spring Data Cassandra provides automatic repository support for the Cassandra database that’s quite similar to—and yet quite different from—what’s offered by Spring Data JPA for relational databases. In addition, Spring Data Cassandra offers mapping annotations to map application domain types to the backing database structures.

Before we explore Cassandra any further, it’s important to understand that although Cassandra shares many similar concepts with relational databases like Oracle and SQL Server, Cassandra isn’t a relational database and is in many ways quite a different beast. I’ll try to explain the idiosyncrasies of Cassandra as they pertain to working with Spring Data. But I encourage you to read Cassandra’s own documentation (<http://cassandra.apache.org/doc/latest/>) for a thorough understanding of what makes Cassandra tick.

Let’s get started by enabling Spring Data Cassandra in the Taco Cloud project.

12.2.1 Enabling Spring Data Cassandra

To get started using Spring Data Cassandra's reactive repository support, you'll need to add the Spring Boot starter dependency for reactive Spring Data Cassandra. There are actually two separate Spring Data Cassandra starter dependencies to choose from.

If you aren't planning to write reactive repositories for Cassandra, you can use the following dependency in your build:

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

This dependency is also available from the Initializr by checking the Cassandra check box.

In this chapter, however, we focus on writing reactive repositories, so you'll want to use the other starter dependency that enables reactive Cassandra repositories:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>
    spring-boot-starter-data-cassandra-reactive
  </artifactId>
</dependency>
```

If you're using the Spring Initializr to create your project, you can get this dependency in your build by checking the Reactive Cassandra check box.

It's important to understand that this dependency is in lieu of the Spring Data JPA starter dependency. Instead of persisting Taco Cloud data to a relational database with JPA, you'll be using Spring Data to persist data to a Cassandra database. Therefore, you'll probably want to remove the Spring Data JPA starter dependency and any relational database dependencies (such as JDBC drivers or the H2 dependency) from the build.

The Spring Data Reactive Cassandra starter dependency brings a handful of dependencies to the project, among which are the Spring Data Cassandra library and Reactor. As a result of those libraries being in the runtime classpath, autoconfiguration for creating reactive Cassandra libraries is triggered. This means you're able to begin writing reactive Cassandra repositories with little explicit configuration.

You'll need to provide a small amount of configuration, though. At the very least, you'll need to configure the name of a key space within which your repositories will operate. To do that, you'll first need to create such a key space.

NOTE In Cassandra, a keyspace is a grouping of tables in a Cassandra node. It's roughly analogous to how tables, views, and constraints are grouped in a relational database.

Although it's possible to configure Spring Data Cassandra to create the key space automatically, it's typically much easier to manually create it yourself (or to use an existing key space). Using the Cassandra CQL (Cassandra Query Language) shell, you can create a key space for the Taco Cloud application with the following create key-space command:

```
cqlsh> create keyspace tacocloud  
... with replication={'class':'SimpleStrategy', 'replication_factor':1}  
... and durable_writes=true;
```

Put simply, this will create a key space named `tacocloud` with simple replication and durable writes. By setting the replication factor to 2, you ask Cassandra to keep one copy of each row. The replication strategy determines how replication is handled. The `SimpleStrategy` replication strategy is fine for single data center use (and for demo code), but you might consider the `NetworkTopologyStrategy` if you have your Cassandra cluster spread across multiple data centers. I refer you to the Cassandra documentation for more details of how replication strategies work and alternative ways of creating key spaces.

Now that you've created a key space, you need to configure the `spring.data.cassandra.keyspace-name` property to tell Spring Data Cassandra to use that key space:

```
spring:  
  data:  
    cassandra:  
      keyspace-name: tacocloud  
      schema-action: recreate-drop-unused
```

Here, you also set the `spring.data.cassandra.schema-action` to `recreate-drop-unused`. This setting is very useful for development purposes because it ensures that any tables and user-defined types will be dropped and recreated every time the application starts. The default value, `none`, takes no action against the schema and is useful in production settings where you'd rather not drop all tables whenever an application starts up.

These are the only properties you'll need for working with a locally running Cassandra database. In addition to these two properties, however, you may wish to set others, depending on how you've configured your Cassandra cluster.

By default, Spring Data Cassandra assumes that Cassandra is running locally and listening on port 9092. If that's not the case, as in a production setting, you may want to set the `spring.data.cassandra.contact-points` and `spring.data.cassandra.port` properties:

```
spring:  
  data:  
    cassandra:  
      keyspace-name: tacocloud  
      contact-points:  
        - casshost-1.tacocloud.com
```

```
- casshost-2.tacocloud.com  
- casshost-3.tacocloud.com  
port: 9043
```

Notice that the `spring.data.cassandra.contact-points` property is where you identify the hostname(s) of Cassandra. A contact point is the host where a Cassandra node is running. By default, it's set to `localhost`, but you can set it to a list of hostnames. It will try each contact point until it's able to connect to one. This is to ensure that there's no single point of failure in the Cassandra cluster and that the application will be able to connect with the cluster through one of the given contact points.

You may also need to specify a username and password for your Cassandra cluster. This can be done by setting the `spring.data.cassandra.username` and `spring.data.cassandra.password` properties:

```
spring:  
  data:  
    cassandra:  
      ...  
      username: tacocloud  
      password: s3cr3tP455w0rd
```

Now that Spring Data Cassandra is enabled and configured in your project, you're almost ready to map your domain types to Cassandra tables and write repositories. But first, let's step back and consider a few basic points of Cassandra data modeling.

12.2.2 Understanding Cassandra data modeling

As I mentioned, Cassandra is quite different from a relational database. Before you can start mapping your domain types to Cassandra tables, it's important to understand a few of the ways that Cassandra data modeling is different from how you might model your data for persistence in a relational database.

These are a few of the most important things to understand about Cassandra data modeling:

- Cassandra tables may have any number of columns, but not all rows will necessarily use all of those columns.
- Cassandra databases are split across multiple partitions. Any row in a given table may be managed by one or more partitions, but it's unlikely that all partitions will have all rows.
- A Cassandra table has two kinds of keys: partition keys and clustering keys. Hash operations are performed on each row's partition key to determine which partition(s) that row will be managed by. Clustering keys determine the order in which the rows are maintained within a partition (not necessarily the order that they may appear in the results of a query).
- Cassandra is highly optimized for read operations. As such, it's common and desirable for tables to be highly denormalized and for data to be duplicated across

multiple tables. (For example, customer information may be kept in a customer table as well as duplicated in a table containing orders placed by customers.)

Suffice it to say that adapting the Taco Cloud domain types to work with Cassandra won't be a matter of simply swapping out a few JPA annotations for Cassandra annotations. You'll have to rethink how you model the data.

12.2.3 Mapping domain types for Cassandra persistence

In chapter 3, you marked up your domain types (Taco, Ingredient, Order, and so on) with annotations provided by the JPA specification. These annotations mapped your domain types as entities to be persisted to a relational database. Although those annotations won't work for Cassandra persistence, Spring Data Cassandra provides its own set of mapping annotations for a similar purpose.

Let's start with the `Ingredient` class, as it's the simplest to map for Cassandra. The new Cassandra-ready `Ingredient` class looks like this:

```
package tacos;
import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Table("ingredients")
public class Ingredient {

    @PrimaryKey
    private final String id;
    private final String name;
    private final Type type;

    public static enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }

}
```

The `Ingredient` class seems to contradict everything I said about just swapping out a few annotations. Rather than annotating the class with `@Entity` as you did for JPA persistence, it's annotated with `@Table` to indicate that ingredients should be persisted to a table named `ingredients`. And rather than annotate the `id` property with `@Id`, this time it's annotated with `@PrimaryKey`. So far, it seems that you're only swapping out a few annotations.

But don't let the `Ingredient` mapping fool you. The `Ingredient` class is one of your simplest domain types. Things get more interesting when you map the `Taco` class for Cassandra persistence.

Listing 12.1 Annotating the Taco class for Cassandra persistence

```

package tacos;
import java.util.Date;
import java.util.List;
import java.util.UUID;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import org.springframework.data.cassandra.core.cql.Ordering;
import org.springframework.data.cassandra.core.cql.PrimaryKeyType;
import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKeyColumn;
import org.springframework.data.cassandra.core.mapping.Table;
import org.springframework.data.rest.core.annotation.RestResource;
import com.datastax.driver.core.utils.UUIDs;
import lombok.Data;

@Data
@RestResource(rel="tacos", path="tacos")
@Table("tacos")                                     ← Persists to
                                                       tacos table
public class Taco {
    @PrimaryKeyColumn(type=PrimaryKeyType.PARTITIONED) ← Defines the
                                                       partition key
    private UUID id = Uuids.timeBased();

    @NotNull
    @Size(min=5, message="Name must be at least 5 characters long")
    private String name;

    @PrimaryKeyColumn(type=PrimaryKeyType.CLUSTERED,
                      ordering=Ordering.DESCENDING) ← Defines the
                                                       clustering key
    private Date createdAt = new Date();

    @Size(min=1, message="You must choose at least 1 ingredient")
    @Column("ingredients")                            ← Maps list to
                                                       ingredients
    private List<IngredientUDT> ingredients;
}

```

As you can see, mapping the Taco class is a bit more involved. As with Ingredient, the @Table annotation is used to identify tacos as the name of the table that tacos should be written to. But that's the only thing similar to Ingredient.

The id property is still your primary key, but it's only one of two primary key columns. More specifically, the id property is annotated with @PrimaryKeyColumn with a type of PrimaryKeyType.PARTITIONED. This specifies that the id property serves as the partition key, used to determine which Cassandra partition(s) each row of taco data will be written to.

You'll also notice that the id property is now a UUID instead of a Long. Although it's not required, properties that hold a generated ID value are commonly of type UUID. Moreover, the UUID is initialized with a time-based UUID value for new Taco objects (but which may be overridden when reading an existing Taco from the database).

A little further down, you see the `createdAt` property that's mapped as another primary key column. But in this case, the `type` attribute of `@PrimaryKeyColumn` is set to `PrimaryKeyType.CLUSTERED`, which designates the `createdAt` property as a clustering key. As mentioned earlier, clustering keys are used to determine the ordering of rows *within a partition*. More specifically, the ordering is set to descending order—therefore, within a given partition, newer rows appear first in the `tacos` table.

Finally, the `ingredients` property is now a `List` of `IngredientUDT` objects instead of a `List` of `Ingredient` objects. As you'll recall, Cassandra tables are highly denormalized and may contain data that's duplicated from other tables. Although the `ingredient` table will serve as the table of record for all available ingredients, the ingredients chosen for a taco will be duplicated in the `ingredients` column. Rather than simply reference one or more rows in the `ingredients` table, the `ingredients` property will contain full data for each chosen ingredient.

But why do you need to introduce a new `IngredientUDT` class? Why can't you just reuse the `Ingredient` class? Put simply, columns that contain collections of data, such as the `ingredients` column, must be collections of native types (integers, strings, and so on) or must be collections of user-defined types.

In Cassandra, user-defined types enable you to declare table columns that are richer than simple native types. Often they're used as a denormalized analog for relational foreign keys. In contrast to foreign keys, which only hold a reference to a row in another table, columns with user-defined types actually carry data that may be copied from a row in another table. In the case of the `ingredients` column in the `tacos` table, it will contain a collection of data structures that define the ingredients themselves.

You can't use the `Ingredient` class as a user-defined type, because the `@Table` annotation has already mapped it as an entity for persistence in Cassandra. Therefore, you must create a new class to define how ingredients will be stored in the `ingredients` column of the `taco` table. `IngredientUDT` (where "UDT" means *user-defined type*) is the class for the job:

```
package tacos;

import org.springframework.data.cassandra.core.mapping.UserDefinedType;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@UserDefinedType("ingredient")
public class IngredientUDT {

    private final String name;
    private final Ingredient.Type type;
}
```

Although `IngredientUDT` looks a lot like `Ingredient`, its mapping requirements are much simpler. It's annotated with `@UserDefinedType` to identify it as a user-defined type in Cassandra. But otherwise, it's a simple class with a few properties.

You'll also note that the `IngredientUDT` class doesn't include an `id` property. Although it could include a copy of the `id` property from the source `Ingredient`, that's not necessary. In fact, the user-defined type may include any properties you wish—it doesn't need to be a one-to-one mapping with any table definition.

I realize that it might be difficult to visualize how data in a user-defined type relates to data that's persisted to a table. Figure 12.1 shows the data model for the entire Taco Cloud database, including user-defined types.

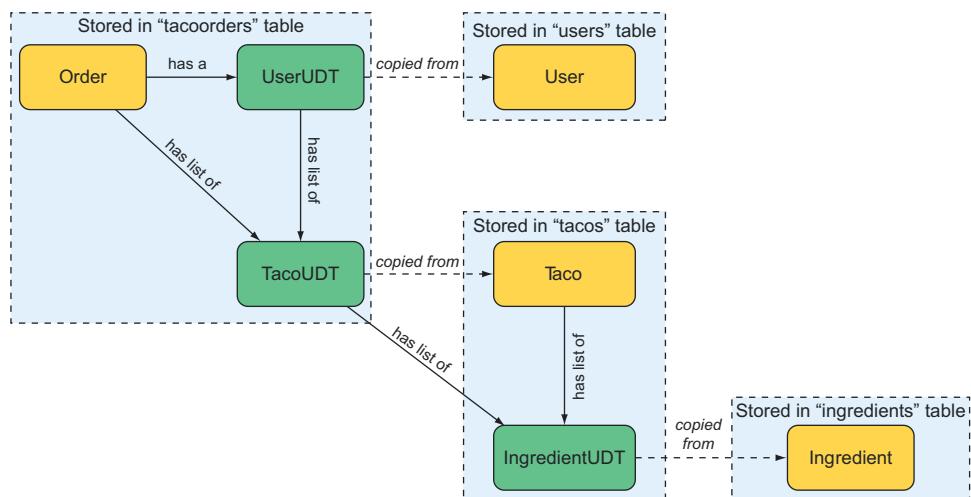


Figure 12.1 Instead of using foreign keys and joins, Cassandra tables are denormalized, with user-defined types containing data copied from related tables.

Specific to the user-defined type that you just created, notice how `Taco` has a list of `IngredientUDT`, which holds data copied from `Ingredient` objects. When a `Taco` is persisted, it's the `Taco` object and the list of `IngredientUDT` that's persisted to the `tacos` table. The list of `IngredientUDT` is persisted entirely within the `ingredients` column.

Another way of looking at this that might help you understand how user-defined types are used is to query the database for rows from the `tacos` table. Using CQL and the `cqlsh` tool that comes with Cassandra, you see the following results:

```
cqlsh:tacocloud> select id, name, createdAt, ingredients from tacos;
id      | name      | createdAt | ingredients
-----+-----+-----+-----
827390...| Carnivore | 2018-04...| [{"name: 'Flour Tortilla', type: 'WRAP'}, {name: 'Carnitas', type: 'PROTEIN'}, {name: 'Sour Cream', type: 'SAUCE'}],
```

```
{name: 'Salsa', type: 'SAUCE'},
{name: 'Cheddar', type: 'CHEESE'}]

(1 rows)
```

As you can see, the `id`, `name`, and `createdat` columns contain simple values. In that regard, they aren't much different than what you'd expect from a similar query against a relational database. But the `ingredients` column is a little different. Because it's defined as containing a collection of the user-defined `ingredient` type (defined by `IngredientUDT`), its value appears as a JSON array filled with JSON objects.

You likely noticed other user-defined types in figure 12.1. You'll certainly be creating some more as you continue mapping your domain to Cassandra tables, including some that will be used by the `Order` class. The next listing shows the `Order` class, modified for Cassandra persistence.

Listing 12.2 Mapping the Order class to a Cassandra `tacoorders` table

```
@Data
@Table("tacoorders")
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

    @PrimaryKey
    private UUID id = UUIDs.timeBased();           ← Maps to
                                                    ← tacoorders table

    private Date placedAt = new Date();             ← Declares the
                                                    ← primary key

    @Column("user")                                ← Maps to the
    private UserUDT user;                          ← user column

    // delivery and credit card properties omitted for brevity's sake

    @Column("tacos")
    private List<TacoUDT> tacos = new ArrayList<>();   ← Maps a list to the
                                                    ← tacos column

    public void addDesign(TacoUDT design) {
        this.tacos.add(design);
    }
}
```

Listing 12.2 purposefully omits many of the properties of `Order` that don't lend themselves to a discussion of Cassandra data modeling. What's left are a few properties and mappings, similar to how `Taco` was defined. `@Table` is used to map `Order` to the `tacoorders` table, much as `@Table` has been used before. In this case, you're unconcerned with ordering, so the `id` property is simply annotated with `@PrimaryKey`, designating it as both a partition key and a clustering key with default ordering.

The `tacos` property is of some interest in that it's a `List<TacoUDT>` instead of a list of `Taco` objects. The relationship between `Order` and `Taco`/`TacoUDT` here is similar to the relationship between `Taco` and `Ingredient`/`IngredientUDT`. That is, rather than joining data from several rows in a separate table through foreign keys, the `Order` table will contain all of the pertinent `taco` data, optimizing the table for quick reads.

Similarly, the `user` property references a `UserUDT` property to be persisted within the `user` column. Again, this is in contrast to the relational database strategy of joining in another table.

As for the `TacoUDT` class, it's quite similar to the `IngredientUDT` class, although it does include a collection that references another user-defined type:

```
@Data  
@UserDefinedType("taco")  
public class TacoUDT {  
  
    private final String name;  
    private final List<IngredientUDT> ingredients;  
  
}
```

The `UserUDT` class is only marginally more interesting in that it has three properties instead of two:

```
@UserDefinedType("user")  
@Data  
public class UserUDT {  
  
    private final String username;  
    private final String fullname;  
    private final String phoneNumber;  
  
}
```

Although it would have been nice to reuse the same domain classes you created in chapter 3, or at most to swap out some JPA annotations for Cassandra annotations, the nature of Cassandra persistence is such that it requires you to rethink how your data is modeled. But now that you've mapped your domain, you're ready to write repositories.

12.2.4 Writing reactive Cassandra repositories

As you saw in chapter 3, writing a repository with Spring Data involves simply declaring an interface that extends one of Spring Data's base repository interfaces and optionally declaring additional query methods for custom queries. As it turns out, writing reactive repositories isn't much different. The primary difference is that you'll extend a different base repository interface, and your methods will deal with reactive publishers such as `Mono` and `Flux`, instead of domain types and collections.

When it comes to writing reactive Cassandra repositories, you have the choice of two base interfaces: `ReactiveCassandraRepository` and `ReactiveCrudRepository`. Which

we choose largely depends on how the repository will be used. `ReactiveCassandraRepository` extends `ReactiveCrudRepository` to offer a few variations of an `insert()` method, which is optimized when the object to be saved is new. Otherwise, `ReactiveCassandraRepository` offers the same operations as `ReactiveCrudRepository`. If you'll be inserting a lot of data, you might choose `ReactiveCassandraRepository`. Otherwise, it's better to stick with `ReactiveCrudRepository`, which is more portable across other database types.

Do my Cassandra repositories have to be reactive?

Although this chapter is about writing reactive repositories with Spring Data, you may be interested to know that you can write non-reactive repositories for Cassandra as well. Rather than extend `ReactiveCrudRepository` or `ReactiveCassandraRepository`, your repository interfaces can extend the non-reactive `CrudRepository` or `CassandraRepository` interfaces. Then your repository methods can simply return Cassandra-annotated domain types and collections of those domain types instead of `Flux` and `Mono`.

If you decide to work with non-reactive repositories, you can also change the starter dependency to `spring-boot-starter-data-cassandra` instead of `spring-boot-starter-data-cassandra-reactive`, although it's not strictly required that you do so.

Revisiting some of the repositories you've already written for the Taco Cloud application, the first thing you should do to make them reactive is to have them extend `ReactiveCrudRepository` or `ReactiveCassandraRepository` instead of `CrudRepository`. For example, consider `IngredientRepository`. Aside from initializing the database with ingredient data, you won't be inserting many new ingredients. Therefore, `IngredientRepository` can extend `ReactiveCrudRepository` as shown here:

```
public interface IngredientRepository
    extends ReactiveCrudRepository<Ingredient, String> { }
```

You never defined any custom query methods in `IngredientRepository`, so there's not much else you need to do to make `IngredientRepository` a reactive repository. But because it now extends `ReactiveCrudRepository`, its methods will deal in terms of `Flux` and `Mono`. For example, the `findAll()` method now returns `Flux<Ingredient>` instead of an `Iterable<Ingredient>`. Consequently, you'll need to be sure that it's being used properly wherever it is being used. The `allIngredients()` method in `IngredientController`, for instance, will need to be rewritten to return a `Flux<Ingredient>`:

```
@GetMapping
public Flux<Ingredient> allIngredients() {
    return repo.findAll();
}
```

The changes to TacoRepository are only subtly more complicated. Instead of extending PagingAndSortingRepository, it will need to extend ReactiveCassandraRepository. And instead of being parameterized for Taco objects with Long ID properties, it will need to work with Taco objects with UUID properties for their IDs:

```
public interface TacoRepository
    extends ReactiveCrudRepository<Taco, UUID> { }
```

Because this new TacoRepository will return Flux<Ingredient> from its findAll() method, you no longer need to worry about it extending PagingAndSortingRepository or working with a page of results. Instead, the recentTacos() method of DesignTacoController will just need to call take() on the returned Flux to limit the number of Taco objects consumed. (In fact, you already made this change to DesignTacoController and its recentTacos() method in section 11.1.2.)

The changes required for OrderRepository are similarly straightforward. Rather than extend CrudRepository, it will now extend ReactiveCassandraRepository:

```
public interface OrderRepository
    extends ReactiveCassandraRepository<Order, UUID> { }
```

Finally, let's look at UserRepository. As you'll recall, UserRepository has a custom query method, findByUsername(). This method adds a little twist to how you must define the repository for Cassandra persistence. Here's what a Cassandra-ready UserRepository interface looks like:

```
public interface UserRepository
    extends ReactiveCassandraRepository<User, UUID> {
    @AllowFiltering
    Mono<User> findByUsername(String username);
}
```

Following suit with all of the other repository interfaces (except IngredientRepository), UserRepository extends ReactiveCassandraRepository. No surprises so far. But its findByUsername() method demands a little bit of extra attention.

First, because this is intended to be a reactive repository, a findByUsername() method that simply returns a User object won't do. You redefine it to return a Mono<User>. Generally speaking, any custom query methods you write in a reactive repository should return either a Mono (if there will be no more than one value returned) or a Flux (if there could be many values returned).

Also, the nature of Cassandra is such that you can't simply query a table with a where clause, like you might do in SQL against a relational database. Cassandra is optimized for reading. But filtering results with a where clause could potentially slow down an otherwise fast query. Even so, querying a table where the results are

filtered by one or more columns is very useful. Therefore, the `@AllowFiltering` annotation makes it possible to filter the results, acting as an opt-in for those cases where it's needed.

In the case of `findByUsername()`, you'd expect a CQL query that looks like this:

```
select * from users where username='some username';
```

Again, that isn't allowed by Cassandra. But when the `@AllowFiltering` annotation is placed on `findByUsername()`, the resulting CQL query looks like this:

```
select * from users where username='some username' allow filtering;
```

The `allow filtering` clause at the end of the query alerts Cassandra that you're aware of the potential impacts to the query's performance and that you need it anyway. In that case, Cassandra will allow the `where` clause and filter the results accordingly.

There's a lot of power in Cassandra, and when it's teamed up with Spring Data and Reactor, you can wield that power in your Spring applications. But let's shift our attention to another database for which reactive repository support is available: MongoDB.

12.3 Writing reactive MongoDB repositories

MongoDB is another well-known NoSQL database. Whereas Cassandra is a row-store database, MongoDB is considered a document database. More specifically, MongoDB stores documents in BSON (Binary JSON) format, which can be queried for and retrieved in a way that's roughly similar to how you might query for data in any other database.

As with Cassandra, it's important to understand that MongoDB isn't a relational database. The way you manage your MongoDB server cluster, as well as how you model your data, requires a different mindset than when working with other kinds of databases.

That said, working with MongoDB and Spring Data isn't dramatically different from how you might use Spring Data for working with JPA or Cassandra. You'll annotate your domain classes with annotations that map the domain type to a document structure. And you'll write repository interfaces that very much follow the same programming model as those you've seen for JPA and Cassandra. Before you can do any of that, though, you must enable Spring Data MongoDB in your project.

12.3.1 Enabling Spring Data MongoDB

To get started with Spring Data MongoDB, you'll need to add the Spring Data MongoDB starter to the project build. Spring Data MongoDB has two separate starters to choose from.

If you're working with non-reactive MongoDB, you'll add the following dependency to the build:

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

This dependency is also available from the Spring Initializr by checking the MongoDB check box. But this chapter is all about writing reactive repositories, so you'll choose the reactive Spring Data MongoDB starter dependency instead:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>
    spring-boot-starter-data-mongodb-reactive
  </artifactId>
</dependency>
```

The reactive Spring Data MongoDB starter can also be added to the build by checking the Reactive MongoDB check box from the Initializr. By adding the starter to the build, autoconfiguration will be triggered to enable Spring Data support for writing automatic repository interfaces, such as those you wrote for JPA in chapter 3 or for Cassandra earlier in this chapter.

By default, Spring Data MongoDB assumes that you have a MongoDB server running locally and listening on port 27017. But for convenience in testing or developing, you can choose to work with an embedded Mongo database instead. To do that, add the Flapdoodle Embedded MongoDB dependency to your build:

```
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo</artifactId>
</dependency>
```

The Flapdoodle embedded database affords you all of the same convenience of working with an in-memory Mongo database as you'd get with H2 when working with relational data. That is, you won't need to have a separate database running, but all data will be wiped clean when you restart the application.

Embedded databases are fine for development and testing, but once you take your application to production, you'll want to be sure you set a few properties to let Spring Data MongoDB know where and how your production Mongo database can be accessed:

```
spring:
  data:
    mongodb:
      host: mongodb.tacocloud.com
      port: 27018
      username: tacocloud
      password: s3cr3tp455w0rd
      database: tacoclouddb
```

Not all of these properties are required, but they're available to help point Spring Data MongoDB in the right direction in the event that your Mongo database isn't running locally. Breaking it down, here's what each property configures:

- `spring.data.mongodb.host`—The hostname where Mongo is running (default: localhost)
- `spring.data.mongodb.port`—The port that the Mongo server is listening on (default: 27017)
- `spring.data.mongodb.username`—The username to use to access a secured Mongo database
- `spring.data.mongodb.password`—The password to use to access a secured Mongo database
- `spring.data.mongodb.database`—The database name (default: test)

Now that you have Spring Data MongoDB enabled in your project, you need to annotate your domain objects for persistence as documents in MongoDB.

12.3.2 Mapping domain types to documents

Spring Data MongoDB offers a handful of annotations that are useful for mapping domain types to document structures to be persisted in MongoDB. Although Spring Data MongoDB provides a half dozen annotations for mapping, only three of them are useful for most common use cases:

- `@Id`—Designates a property as the document ID (from Spring Data Commons)
- `@Document`—Declares a domain type as a document to be persisted to MongoDB
- `@Field`—Specifies the field name (and optionally the order) for storing a property in the persisted document

Of those three annotations, only the `@Id` and `@Document` annotations are strictly required. Unless you specify otherwise, properties that aren't annotated with `@Field` will assume a field name equal to the property name.

Applying these annotations to the `Ingredient` class, you get the following:

```
package tacos;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Document
public class Ingredient {

    @Id
    private final String id;
```

```

private final String name;
private final Type type;

public static enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
}

}

```

As you can see, you place the `@Document` annotation at the class level to indicate that `Ingredient` is a document entity that can be written to and read from a Mongo database. By default, the collection name (the Mongo analog to a relational database table) is based on the class name, with the first letter lowercased. Because you haven't specified otherwise, `Ingredient` objects will be persisted to a collection named `ingredient`. But you can change that by setting the `collection` attribute of `@Document`:

```

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Document(collection="ingredients")
public class Ingredient {
...
}

```

You'll also notice that the `id` property has been annotated with `@Id`. This designates the property as being the ID of the persisted document. You can use `@Id` on any property whose type is `Serializable`, including `String` and `Long`. In this case, you're already using the `String`-defined `id` property as a natural identifier, so there's no need to change it to any other type.

So far, so good. But you'll recall from earlier in this chapter that `Ingredient` was the easy domain type to map for Cassandra. The other domain types, such as `Taco`, were a bit more challenging. Let's look at how you can map the `Taco` class to see what surprises it might hold.

As with any domain-to-document mapping for MongoDB, you'll certainly need to annotate `Taco` with `@Document`. And you'll also need to designate an ID property with `@Id`. Doing so yields the following `Taco` class annotated for MongoDB persistence:

```

@Data
@RestResource(rel="tacos", path="tacos")
@Document
public class Taco {

    @Id
    private String id;

    @NotNull
    @Size(min=5, message="Name must be at least 5 characters long")
    private String name;

    private Date createdAt = new Date();
}

```

```

@Size(min=1, message="You must choose at least 1 ingredient")
private List<Ingredient> ingredients;

}

```

Believe it or not, that's it! The challenges of dealing with two different primary key fields and referencing user-defined types were specific to Cassandra. For MongoDB, the Taco mapping is much simpler.

Even so, there are a few interesting things to point out in Taco. First, notice that the `id` property has been changed to be a `String` (as opposed to a `Long` in the JPA version or a `UUID` in the Cassandra version). As I said earlier, `@Id` can be applied to any `Serializable` type. But if you choose to use a `String` property as the ID, you get the benefit of Mongo automatically assigning a value to it when it's saved (assuming that it's `null`). By choosing `String`, you get a database-managed ID assignment and needn't worry about setting that property manually.

Also, take a look at the `ingredients` property. Notice that it's a `List<Ingredient>`, just like it was in the JPA version from chapter 3. But unlike the JPA version, the list isn't stored in a separate MongoDB collection. Much like its Cassandra counterpart, the list of ingredients is stored directly, denormalized, in the `taco` document. But unlike the Cassandra implementation, you don't need to make up a user-defined type—MongoDB is happy to use any type here, whether it's another `@Document`-annotated type or just a POJO.

It certainly is a relief to see that mapping Taco for document persistence is easy. Will that ease of mapping carry over to the `Order` domain class? Take a look at the following MongoDB-annotated `Order` class to see for yourself:

```

@Data
@Document
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String id;

    private Date placedAt = new Date();

    @Field("customer")
    private User user;

    // other properties omitted for brevity's sake

    private List<Taco> tacos = new ArrayList<>();

    public void addDesign(Taco design) {
        this.tacos.add(design);
    }

}

```

For brevity's sake, I've snipped out the various delivery and credit card fields. But from what's left, it's clear that all you need is `@Document` and `@Id`, as with the other domain types. Even so, you annotate the `user` property with `@Field` to specify that it be stored as `customer` in the persisted document.

By now, it shouldn't be surprising that mapping the `User` domain class for MongoDB persistence should be just as easy:

```
@Data  
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)  
@RequiredArgsConstructor  
@Document  
public class User implements UserDetails {  
  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    private String id;  
  
    private final String username;  
  
    private final String password;  
    private final String fullname;  
    private final String street;  
    private final String city;  
    private final String state;  
    private final String zip;  
    private final String phoneNumber;  
  
    // UserDetails method omitted for brevity's sake  
  
}
```

Although there are some more-advanced and unusual use cases that require additional mapping, you'll find that for most cases, `@Document` and `@Id`, along with an occasional `@Field`, are sufficient for MongoDB mapping. They certainly do the job for the Taco Cloud domain types.

All that's left is to write the repository interfaces.

12.3.3 Writing reactive MongoDB repository interfaces

Spring Data MongoDB offers automatic repository support similar to what's provided by Spring Data JPA and Spring Data Cassandra. When it comes to writing reactive repositories for MongoDB, you have a choice between `ReactiveCrudRepository` and `ReactiveMongoRepository`. The key difference is that `ReactiveMongoRepository` provides a handful of special `insert()` methods that are optimized for persisting new documents, whereas `ReactiveCrudRepository` relies on `save()` methods for new and existing documents.

What about non-reactive MongoDB repositories?

The focus of this chapter is on writing reactive repositories with Spring Data. But if for some reason you wish to work with non-reactive repositories, you can do so by simply having your repository interfaces extend `CrudRepository` or `MongoRepository` instead of `ReactiveCrudRepository` or `ReactiveMongoRepository`. Then you can have the repository methods return Mongo-annotated domain types and collections of those domain types.

It's not strictly required that you do so, but you can also choose to change the `spring-boot-starter-data-mongodb-reactive` dependency to `spring-boot-starter-data-mongodb`.

You'll start by defining a repository for persisting `Ingredient` objects as documents. You won't be creating ingredient documents frequently, or at all, after the database is initialized. Therefore, the optimizations offered by `ReactiveMongoRepository` won't be as helpful. You can write `IngredientRepository` to extend `ReactiveCrudRepository`:

```
package tacos.data;
import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import org.springframework.web.bind.annotation.CrossOrigin;
import tacos.Ingredient;

@CrossOrigin(origins = "*")
public interface IngredientRepository
    extends ReactiveCrudRepository<Ingredient, String> {
}
```

Wait a minute! That looks *identical* to the `IngredientRepository` interface you wrote in section 12.2.4 for Cassandra! Indeed, it's the same interface, with no changes. This highlights one of the benefits of extending `ReactiveCrudRepository`—it's more portable across various database types and works equally well for MongoDB as for Cassandra.

Because it's a reactive repository, its methods deal in terms of `Flux` and `Mono` rather than raw domain types and collections of those domain types. The `findAll()` method, for instance, will return `Flux<Ingredient>` instead of `Iterable<Ingredient>`. Likewise, `findById()` will return `Mono<Ingredient>` instead of `Optional<Ingredient>`. As a result, this reactive repository could be part of an end-to-end reactive flow.

Now let's try defining a repository for persisting `Taco` objects as documents in MongoDB. Unlike ingredient documents, you'll be creating `taco` documents rather frequently. Thus, the optimized `insert()` methods from `ReactiveMongoRepository` might prove valuable. Here's your new MongoDB-ready `TacoRepository` interface:

```
package tacos.data;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import reactor.core.publisher(Flux;
import tacos.Taco;
```

```
public interface TacoRepository
    extends ReactiveMongoRepository<Taco, String> {

    Flux<Taco> findByOrderByCreatedAtDesc();

}
```

The only drawback of using `ReactiveMongoRepository` as opposed to `ReactiveCrudRepository` is that it's very specific to MongoDB and not portable to other databases. In your projects, you'll need to decide if that trade-off is worth it or not. If you don't anticipate switching to a different database at some point, it's safe enough to choose `ReactiveMongoRepository` and benefit from the insertion optimizations.

Notice that you introduce a new method in `TacoRepository`. This method is to support the use case of presenting a list of recently created tacos. In the JPA version of this repository, you achieved that by extending `PagingAndSortingRepository`. But `PagingAndSortingRepository` doesn't make much sense (especially the paging part of it) in a reactive repository. In the Cassandra version, sorting was determined by the clustering key in the table definition, so you didn't have anything special in the repository to support fetching recent taco creations.

But for MongoDB, you'd like to be able to fetch the most recently created tacos. Despite its odd name, the `findByOrderByCreatedAtDesc()` method follows the custom query method-naming convention. It says that you're finding a `Taco` object by, well, by nothing. You don't specify any properties that must match. Then you tell it to order the results by the `createdAt` property in descending order.

The reason to name it with an empty `By` clause is to avoid a misinterpretation of the method name, given that there's another `By` in the method name. Had you named it `findAllOrderByCreatedAtDesc()`, the `AllOrder` portion of the name would've been ignored, and Spring Data would try to find tacos by matching against a `createdAt-Desc` property. Because no such property exists, the application would fail to start, with an error.

Because `findByOrderByCreatedAtDesc()` returns a `Flux<Taco>`, you needn't worry about paging. Instead, you can simply apply the `take()` operation to take only the first dozen `Taco` objects published in the `Flux` returned. For example, your controller that displays the recently created tacos could make a call to `findByOrderByCreatedAtDesc()` like this:

```
Flux<Taco> recents = repo.findByOrderByCreatedAtDesc()
    .take(12);
```

The resulting `Flux` would only ever have, at most, 12 `Taco` items published.

Moving on to the `OrderRepository` interface, you can see that it's straightforward:

```
package tacos.data;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import reactor.core.publisher.Flux;
import tacos.Order;
```

```
public interface OrderRepository
    extends ReactiveMongoRepository<Order, String> {
}
```

You'll be frequently creating `Order` documents, so `OrderRepository` extends `ReactiveMongoRepository` to gain the optimizations afforded in its `insert()` methods. Otherwise, there's nothing terribly special about this repository, compared to some of the other repositories you've defined thus far.

Finally, let's take a look at the repository that will persist `User` objects as documents:

```
package tacos.data;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import reactor.core.publisher.Mono;
import tacos.User;

public interface UserRepository
    extends ReactiveMongoRepository<User, String> {

    Mono<User> findByUsername(String username);

}
```

By now, there should be nothing terribly surprising about this repository interface. Like the others, it extends `ReactiveMongoRepository` (although it could have also extended `ReactiveCrudRepository`). The only thing unique is the addition of a `findByUsername()` method, which you added in chapter 4 to support authentication against this repository. Here, it's been tweaked to return a `Mono<User>` instead of a raw `User` object.

Summary

- Spring Data supports reactive repositories for Cassandra, MongoDB, Couchbase, and Redis databases.
- Spring Data's reactive repositories follow the same programming model as non-reactive repositories, except that they deal in terms of reactive publishers such as `Flux` and `Mono`.
- Non-reactive repositories (such as JPA repositories) can be adapted to work with `Mono` and `Flux`, but they ultimately still block while data is saved and fetched.
- Working with nonrelational databases demands an understanding of how to model data appropriately for how the database ultimately stores the data.

Part 4

Cloud-native Spring

P

art 4 breaks down the monolithic application model, introducing you to Spring Cloud and microservice development. In chapter 13, after a brief introduction to microservices, we'll dive into service discovery, using Spring with Netflix's Eureka service registry to both register and discover Spring-based microservices. Chapter 14 explores centralized configuration using Spring Cloud's Config Server, a service that provides centralized configuration for all services in a given application. In chapter 15, you'll see how to apply the circuit breaker pattern to make services more resilient to failure with Netflix Hystrix.

13

Discovering services

This chapter covers

- Thinking in microservices
- Creating a service registry
- Registering and discovering services

Have you ever watched *Finding Nemo*? In that movie, Marlin (a clown fish) and Dory (a blue tang fish) are trying to get to Sydney, Australia to find Marlin's missing son, Nemo. Along the way, they encounter a school of moonfish. For fun, the moonfish arrange themselves into several shapes—a swordfish, an octopus, and they even mock Marlin by arranging themselves to look like him. When Dory asks them if they know how to get to Sydney, they form the shape of the Sydney Opera House and then change into an arrow pointing toward the east Australian current.

Although the movie doesn't delve into the lives of any particular moonfish, it can be assumed that each of them is an individual, independent of the other moonfish. Each has its own scales, fins, gills, eyes, internal organs, and (as far as we know) their own hopes and dreams. Even so, they still work together to form those fun shapes and help Marlin and Dory make their way to Australia.

This chapter is the first of a handful of chapters that discuss how to develop applications that are composed of moonfish. That is, you’ll see how to develop with *microservices*—small, independent applications that work together to provide the functionality of a complete application.

More specifically, you’re going to see how to use a few of the most useful components in the Spring Cloud portfolio, including configuration management, failure tolerance, and the subject of this chapter—service discovery. But before we get started, let’s take a quick, high-level look at what it means to develop with microservices and the benefits they offer.

13.1 Thinking in microservices

Up to this point, you’ve developed the Taco Cloud application as a single application that builds into a single, deployable JAR or WAR file. A single, deployable file seemed like a natural move. After all, that’s exactly how most applications have been built for decades. Even when the application is broken down into a multi-module build, you still end up with a single JAR or WAR file that you push into production.

This is certainly the most obvious way to build small, simple applications. But the funny thing about small applications is that they tend to grow. It’s easy enough to drop more code into the project when a new feature is needed. Before you know it, you have a complex, monolithic application that has a mind of its own. Like the Mogwai in the movie *Gremlins*, if you keep feeding it, it’ll eventually become a monster that turns against you.¹

Monolithic applications are deceptively simple, but they present a few challenges:

- *Monoliths are difficult to reason about*—The bigger the codebase gets, the harder it is to comprehend each component’s role in the whole application.
- *Monoliths are more difficult to test*—As the application grows, comprehensive integration and acceptance testing gets more complicated.
- *Monoliths are more prone to library conflicts*—One feature may require a dependency that’s incompatible with the dependency required by another.
- *Monoliths scale inefficiently*—If you need to deploy the application to more hardware for scaling purposes, you must deploy the entire application to more servers—even if it’s only a small fraction of the application that requires scaling.
- *Technology decisions for a monolith are made for the entire monolith*—When you choose a language, runtime platform, framework, or library for your application, you choose it for the entire application, even if the choice is made to only support a single use case.
- *Monoliths require a great deal of ceremony to get to production*—It would seem that when an application has only a single deployment unit, it would be easier to get

¹ Yes, I realize that the timing of when you feed Mogwai—after midnight—was the real issue in the movie. No analogy is perfect.

into production. In reality, however, the size and complexity of monolithic applications generally require a more rigid development process and a more thorough testing cycle to ensure that what's deployed is of high quality and doesn't introduce bugs.

In the past few years, microservice architecture has risen to address these challenges. In simple terms, *microservice architecture* is a way of factoring an application into small-scale, miniature applications that are independently developed and deployed. These microservices coordinate with each other to provide the functionality of a greater application. In contrast to monolithic application architecture, microservice architecture has these traits:

- *Microservices can be easily understood*—Each microservice has a small, finite contract with other microservices in the greater application. As a result, microservices are more focused in purpose and, therefore, easier to understand as a unit.
- *Microservices are easier to test*—The smaller something is, the easier it is to test. This is certainly evident when you consider unit testing versus integration testing versus acceptance testing. That also applies when testing microservices versus monolithic applications.
- *Microservices are unlikely to suffer from library incompatibilities*—Because each microservice has its own set of build dependencies that isn't shared with other microservices, it's less likely that there'll be library conflicts.
- *Microservices scale independently*—If any given microservice needs more horsepower, then the memory allotment and/or the number of instances can be scaled up without impacting the memory or instance count of other microservices in the greater application.
- *Technology choices can be made differently for each microservice*—Entirely different decisions can be made with regard to the language, platform, framework, and library choices for each microservice. In fact, it's entirely reasonable for one microservice written in Java to coordinate with another microservice written in C#.²
- *Microservices can be published to production more frequently*—Even though a microservice-architected application is made up of many microservices, each one can be deployed without requiring that any of the other microservices also be deployed. And because they're smaller, more focused, and easier to test, there's less ceremony in taking a microservice into production. The time between having an idea and seeing it in production can potentially be measured in minutes and hours, instead of weeks and months.

Microservices certainly seem to make things a lot easier. But to be fair, microservice architecture isn't exactly a free lunch. Microservice architecture is a distributed

² We'll focus on microservices written with Java and Spring. But if you're interested in how to write microservices in .NET that work with Spring Cloud Services, have a look at Steeltoe (<http://steeltoe.io/>).

architecture that brings its own challenges to the table, including network latency. As you make the move to microservice architecture, you should keep this in mind as many remote calls can add up and slow down an application.

You should also consider whether it even makes sense to architect your application in microservices. Not all applications require or benefit from such an architecture. If the application is relatively small or simple, perhaps it's best to leave it as a monolith ... for now. As it grows, you can begin to consider breaking it into microservices.

There's a lot of thought that goes into developing cloud-native, microservice-architected applications. This chapter and the next few chapters will focus primarily on the technology afforded by Spring Cloud to develop applications that are made up of microservices. But if you're interested in digging deeper into the design and thought process around cloud-native applications, may I suggest that you read *Cloud Native* by Cornelia Davis (Manning, 2019, www.manning.com/books/cloud-native).

Another common challenge faced by microservice architecture is how each service even knows about the other services it coordinates with. That's precisely the topic of this chapter. Without further delay, let's see how to set up a service registry with Spring Cloud.

13.2 **Setting up a service registry**

Spring Cloud is a rather large umbrella project, made up of several separate sub-projects that each enables microservice development in some way. One of those subprojects is Spring Cloud Netflix, which offers several components from the Netflix open source portfolio with a Spring twist. Among those components is Eureka, the Netflix service registry.

THE NAKED TRUTH CONCERNING EUREKA

The word *Eureka* is an exclamation of joy when one finds or discovers something. This makes it a fitting name for the service registries that will be used by microservices to discover each other.

Legend says that Eureka was first uttered by Greek physicist Archimedes when, on discovering the principle of buoyant force while sitting in a bath, he leapt out of the bath and ran home naked shouting “Eureka!”

There's some debate whether or not Archimedes actually ran home in the buff shouting “Eureka!” But the story is amusing, nonetheless. Regardless, we're able to work with the Eureka service registry fully-clothed.

Eureka acts as a central registry for all services in a microservice application. Eureka itself can be thought of as a microservice—one whose purpose in the greater application is to help the other services discover each other.

Because of its role in a microservice application, it's probably best to set up a Eureka service registry before creating any of the services that register with it. To understand how Eureka works, consider the flow as described in figure 13.1.

When a service instance starts, it'll register itself by name with Eureka. In figure 13.1, the service name is `some-service`. There may be multiple equivalent instances of `some-service`, but all of them register with Eureka with the same name.

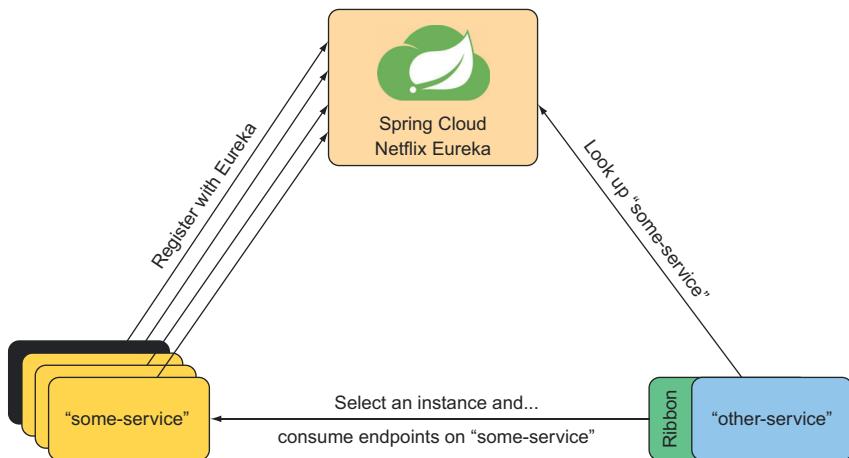


Figure 13.1 Services register with the Eureka service registry so that other services can discover and consume them.

At some point, another service (named other-service in figure 13.1) needs to consume endpoints on some-service. Rather than hard coding other-service with specific host and port information for some-service, other-service only knows to look up some-service from Eureka by its name. Eureka replies with information for all instances of some-service that it knows about.

Now other-service needs to make a decision. Which instance of some-service will it use? If they're all equivalent, then it doesn't matter much. But to avoid any given instance being chosen every time, it's best to apply some client-side, load-balancing algorithm to spread the requests around. That's where another Netflix project—Ribbon—comes into play.

Although other-service could be solely responsible for both looking up and choosing an instance of some-service, it relies on Ribbon instead. Ribbon is a client-side load balancer that makes the choice on behalf of other-service. Once Ribbon has made its choice, all that's left is for other-service to make requests to the instance that Ribbon chooses.

WHY A CLIENT-SIDE LOAD BALANCER?

Often, load balancers are thought of as a single centralized service that handles all requests and distributes them across many instances of the intended target. In contrast, Ribbon is a client-side load balancer that's local to each client making the requests.

As a client-side load balancer, Ribbon has several benefits over a centralized load balancer. Because there's one load balancer local to each client, the load balancer naturally scales proportional to the number of clients. Furthermore, each load balancer can be configured to employ a load-balancing algorithm best suited for each client, rather than apply the same configuration for all services.

If this seems complex, don't worry. As you'll soon see, most of this is handled automatically and transparently. But before you can register and consume services, you need to enable a Eureka server.

To get started with Spring Cloud and Eureka, you'll need to create a brand new project for the Eureka server itself. The easiest way to get started is with the Spring Initializr. Name the project whatever you want, although I'm inclined to name it service-registry. When it comes time to select starter dependencies, there's only one dependency you'll need: the one whose checkbox is labeled Eureka Server. After creating the new project, Initializr gives you a project whose pom.xml file contains the following dependency:

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

You'll also see a property named `spring-cloud.version` and a `<dependencyManagement>` section in the pom.xml file that specifies the Spring Cloud release train version. When I created my service-registry, it referenced the first service release (SR1) of the Finchley release train:

```
<properties>
  ...
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>

...

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

If you'd like to use a different version of Spring Cloud, you only need to change the `spring-cloud.version` property to the desired version.

With the Eureka starter dependency in the build, there's only one other thing you must do to enable the Eureka server. Open the application's main bootstrap class and annotate it with `@EnableEurekaServer`:

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {
```

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

And that's it! If you start the application, you'll have a Eureka service registry running and listening on port 8080. If you point your web browser to <http://localhost:8080>, you'll see the web interface shown in figure 13.2.

The Eureka dashboard is informative, telling you (among other things) what service instances are registered with Eureka. You'll find yourself viewing this UI frequently as

The screenshot shows the Spring Eureka web dashboard. At the top, there's a header with the Eureka logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section displays environment details (Environment: test, Data center: default), current time (2018-04-15T16:30:17 -0600), uptime (00:00), lease expiration settings (false), and renew thresholds (0). The 'DS Replicas' section shows a table with one row: 'localhost'. The 'General Info' section lists various system metrics like memory usage and server uptime. The 'Instance Info' section shows the IP address (10.0.0.94) and status (UP).

Name	Value
total-avail-memory	634mb
environment	test
num-of-cpus	8
current-memory-usage	499mb (78%)
server-uptime	00:00
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/ ,
available-replicas	

Name	Value
ipAddr	10.0.0.94
status	UP

Figure 13.2 The Eureka web dashboard

you register services to ensure that they’re registered as expected. At this point, no services are registered, thus the message No Instances Available.

Eureka also exposes a REST API, through which services register themselves and discover other services. You likely won’t use the REST API directly, but you might find the /eureka/apps endpoint interesting. It lists in detail all of the service instances in the registry. At this point, with no service instances registered, the response you’ll get follows. We’ll revisit this endpoint a little later in the chapter as you start registering services:

```
<applications>
  <versions_delta>1</versions_delta>
  <apps_hashcode></apps_hashcode>
</applications>
```

You may have noticed that Eureka logs several exceptions in its log every 30 s or so. Don’t worry! Eureka is running and will work as expected. But those exceptions are indicative of the fact that you’ve not fully configured the service registry yet. Let’s add a few configuration properties to make those exceptions disappear.

13.2.1 Configuring Eureka

Eureka doesn’t like working alone. Eureka believes in the concept of safety in numbers and wants to be part of a cluster of Eureka servers. If there’s more than one Eureka server, then there’s no single point of failure should one of those servers run into trouble. Therefore, Eureka’s default behavior is to attempt to make friends with other Eureka servers. It’ll try to fetch the service registry from other Eureka servers and even register itself as a service with other Eureka servers.

High availability of Eureka is desirable in a production setting. For development, however, it’s inconvenient and unnecessary to fire up more than one Eureka server. A single, lonely Eureka server is sufficient for development purposes. But unless you configure the Eureka server properly, it’ll whine about its loneliness incessantly, every 30 s, in the form of exceptions in the log file. That’s because every 30 s, Eureka is trying to get in touch with another Eureka server to register itself and to share registry information.

What you need to do is configure Eureka to accept a solitary existence. To do this, you’ll need to set a handful of configuration properties as shown in the following snippet from application.yml:

```
eureka:
  instance:
    hostname: localhost
  client:
    fetch-registry: false
    register-with-eureka: false
    service-url:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka
```

First, you set the `eureka.instance.hostname` property to `localhost`. This tells Eureka what host it's running on. It's optional, but if you don't specify it, then Eureka attempts to determine its host from environment variables. Explicitly setting this property gives you more certainty of what its value will be.

The next two properties, `eureka.client.fetch-registry` and `eureka.client.register-with-eureka`, are properties that you might set on another microservice to tell them how they should interact with Eureka. But recall that Eureka is also a microservice, so these properties can be used with a Eureka server to tell it how it should interact with other Eureka servers.

The default value for both of these properties is `true`, indicating that Eureka should fetch the registry from other Eureka instances, and it should register itself as a service with the other Eureka servers. Because there aren't other Eureka servers when in development mode, you're setting them to `false` so that Eureka won't try to reach out to other Eureka servers.

Finally, you set the `eureka.client.service-url` property. This property contains a map of zone names to one or more URLs of Eureka servers in that zone. The map key of `defaultZone` is a special zone name, which is the zone that should be used if the client (in this case, Eureka itself) hasn't specified a desired zone. Because this is the only Eureka instance, the URL mapped to the default zone is for the Eureka server itself, using placeholder variables populated from other properties.

SPECIFYING EUREKA'S SERVER PORT

Although it's optional, you'll probably want to override the default server port. Whereas Eureka is perfectly happy to listen on port 8080, you'll run several applications (microservices) simultaneously on the local machine as you develop the code, and the applications can't all listen on port 8080. Therefore, it's generally a good idea to set the `server.port` property for local development purposes:

```
server:  
  port: 8761
```

Here you're setting it to port 8761, which is the default port that the Eureka client (which we'll discuss in section 13.3) listens on.

DISABLING SELF-PERSERVATION MODE

One other property that you may consider setting is `eureka.server.enable-self-preservation`. If you ever start the Eureka server and let it sit idle for more than a minute or so, you may see a scary message in the Eureka UI that looks like figure 13.3.

In spite of the red lettering and all uppercase text, this message isn't nearly as serious as it sounds. Eureka expects service instances to register themselves and to continue to send registration renewal requests every 30 s. Normally, if Eureka doesn't receive a renewal from a service for three renewal periods (or 90 s), it deregisters that instance. In this case, Eureka assumes there's a network problem, enters self-preservation mode, and won't deregister service instances.

The screenshot shows the Spring Eureka dashboard. At the top, there's a navigation bar with the Spring logo and the word "Eureka". On the right side of the bar are links for "HOME" and "LAST 1000 SINCE STARTUP". Below the bar, the title "System Status" is displayed. There are two tables: one for environment details and one for system metrics. The environment table shows "Environment" as "test" and "Data center" as "default". The metrics table shows "Current time" as "2018-04-15T17:17:51 -0600", "Uptime" as "00:22", "Lease expiration enabled" as "true", "Renews threshold" as "0", and "Renews (last min)" as "0". A red message box at the bottom states: "RENEWALS ARE LESSER THAN THE THRESHOLD. THE SELF PRESERVATION MODE IS TURNED OFF.THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS."

Environment	test
Data center	default

Current time	2018-04-15T17:17:51 -0600
Uptime	00:22
Lease expiration enabled	true
Renews threshold	0
Renews (last min)	0

Figure 13.3 When in self-preservation mode, Eureka displays this message in its dashboard.

Self-preservation mode is actually a good thing in a production setting, preventing the deregistration of active services when some network hiccup has stopped the renewal request from making its way to Eureka. But it can be quite alarming when you’re first starting up and haven’t registered any services yet. You can disable self-preservation mode by setting the `eureka.server.enable-self-preservation` property to `false`:

```
eureka:
...
server:
  enable-self-preservation: false
```

This property is useful in a development environment where it’s likely that for a number of reasons, Eureka may not receive renewal requests. In those cases where you may be bringing service instances up and down frequently, self-preservation mode results in registry entries for stopped services being retained, creating problems when another service tries to consume the service that’s long gone. Disabling self-preservation mode will prevent strange behavior like that. The tradeoff, however, is that you’ve traded one scary red-letter message for another (figure 13.4).

THE SELF PRESERVATION MODE IS TURNED OFF.THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.

Figure 13.4 When self-preservation mode is disabled, you’ll get a different message reminding you that self-preservation mode is disabled.

Even if you decide to disable self-preservation mode for development, you should leave it enabled when you go into production.

13.2.2 Scaling Eureka

Even though a single Eureka instance is more convenient in development, you'll probably want to have at least two Eureka instances for high-availability purposes when you take the application to production.

PRODUCTION-READY SPRING CLOUD SERVICES

There's a lot to consider when deploying microservices into a production environment. High availability and security of the Eureka server are a few of the concerns that aren't as important at development time, but are critical in production. If you're a Pivotal Cloud Foundry or Pivotal Web Services customer, then you can let someone else worry about those things.

Spring Cloud Services offers a production-ready implementation of Eureka, as well as a configuration server and a circuit breaker dashboard. All you need to do is provision a p-service-registry service from the marketplace and then bind your microservices to that service. For the configuration server and the circuit breaker dashboard (which we'll discuss in the next couple of chapters), the marketplace names are p-config-server and p-circuit-breaker-dashboard.

The easiest, most straightforward way to configure two (or more) Eureka instances is to use Spring profiles in the application.yml file and then start the Eureka server twice, once for each profile. For example, the configuration in the next listing shows how you might configure two Eureka servers that act as peers to each other.

Listing 13.1 Configuring Eureka for two peers using Spring profiles

```
eureka:
  client:
    service-url:
      defaultZone: http://${other.eureka.host}:${other.eureka.port}/eureka

  ---
spring:
  profiles: eureka-1
  application:
    name: eureka-1

  server:
    port: 8761

  eureka:
    instance:
      hostname: eureka1.tacocloud.com

  other:
    eureka:
      host: eureka2.tacocloud.com
      port: 8761

  ---
spring:
```

```

profiles: eureka-2
application:
  name: eureka-2

server:
  port: 8762

eureka:
  instance:
    hostname: eureka2.tacocloud.com

other:
  eureka:
    host: eureka1.tacocloud.com
    port: 8762

```

In the default profile (at the top of listing 13.1), you set `eureka.client.serviceUrl.defaultZone` to take advantage of placeholder variables that you set in each of the profile-specific configurations.

After the default profile, you configure two profiles, one named `eureka-1` and the other named `eureka-2`. Each profile specifies its own port and `eureka.instance.hostname` for its own configuration needs. But then you set `other.eureka.host` and `other.eureka.port`, two contrived properties, in each profile to reference the other Eureka instance. There's nothing about these properties that's specific to the framework, but these properties are what get referenced by the placeholders in the default profile.

Notice that you're not setting `eureka.client.fetch-registry` or `eureka.client.register-with-eureka`. The default value of `true` ensures that each Eureka server registers itself and fetches registry information from the other Eureka server.

Now you have a Eureka service registry up and running. But at this point, it's essentially a phonebook with empty pages that nobody ever looks at. Until services start registering themselves in the service registry and other services look up those services and call them, it's all for naught. Let's see how to enable some microservices as Eureka clients.

13.3 **Registering and discovering services**

A Eureka service registry is useless unless services register themselves. If your services are going to be discovered and consumed by other services, then you need to enable them as clients of the service registry. To enable an application (any application, but presumably a microservice) as a service registry client, the least you must do is add the Eureka client dependency to the service application's build:

```

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

```

As with the Eureka server starter dependency, you'll also need the Spring Cloud version property set for Spring Cloud's dependency management:

```
<properties>
  ...
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>

...

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

You can add these entries to your service application's pom.xml file manually, but the easier way to get them is to select the Eureka Discovery dependency from the Spring Initializr's selection of checkboxes.

The Eureka client starter dependency adds all you need for discovering services via Eureka, including Eureka's client-side library, as well as the Ribbon load balancer. By doing nothing more than adding this dependency, you'll enable your application as a client of the Eureka service registry. When the application starts, it attempts to contact a Eureka server running locally and listening on port 8761 to register itself under the name UNKNOWN.

13.3.1 Configuring Eureka client properties

Although the default location of the Eureka server is fine for development purposes, you'll most certainly want to override it once your services are deployed outside of localhost. What's more, that default service name of UNKNOWN is a horrible choice ... but truthfully, any default choice would be equally bad, as all services would have the same name.

Changing the name under which the service is registered in Eureka is as easy as setting the `spring.application.name` property. For example, if you're registering a service that handles all operations that involve dealing with taco ingredients, you might register it as `ingredient-service`. In `application.yml` that would look like this:

```
spring:
  application:
    name: ingredient-service
```

As a result of setting this property, the service can be looked up by the name `ingredient-service`. What's more, if you were to fire up multiple instances of the ingredient service, they'd all appear under the same name, effectively scaling the service up to

multiple, presumably equivalent, instances that a consuming service can choose from. When you view the Eureka dashboard, this service appears as shown in figure 13.5.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
INGREDIENT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.3:ingredient-service

Figure 13.5 The ingredient service as it appears in Eureka’s dashboard

You’ll find as you continue to work with Spring Cloud that the `spring.application.name` property is one of the most important properties that you’ll ever set. It determines the registry name in Eureka. And in the next chapter, you’ll see that it identifies the application to the configuration service for managing application-specific configurations. Other Spring Cloud projects, such as Spring Cloud Task (ephemeral microservices) and Spring Cloud Sleuth (distributed tracing), also rely on the `spring.application.name` property to identify the service.

As you’ve learned from the first chapter, all Spring MVC and Spring WebFlux applications are listening on port 8080 by default. Because you’ll only be looking up services through Eureka, it doesn’t matter what port they’re listening on—Eureka knows what port they’re on. Therefore, to avoid potential port conflicts when running locally, you can set the port to 0:

```
server:
  port: 0
```

NOTE: Setting the port to 0 results in the application starting on a randomly chosen available port.

Now, regarding the location of the Eureka servers. By default, Eureka clients assume that Eureka is listening on localhost (port 8761). That’s great for development, but in production that most certainly isn’t the case. Therefore, you’ll need to specify the location of your Eureka server(s). That’s accomplished the same way you achieved it for the Eureka server itself, with the `eureka.client.service-url` property:

```
eureka:
  client:
    service-url:
      defaultZone: http://eureka1.tacocloud.com:8761/eureka/
```

This configures the client to register with the Eureka server listening on host `eureka1.tacocloud.com` (port 8761). That’s fine, so long as that Eureka server is in working order. But if that Eureka server is down for any reason, then the service fails to

register. To avoid registration failure, it's best to configure your service with two or more Eureka servers:

```
eureka:  
  client:  
    service-url:  
      defaultZone: http://eureka1.tacocloud.com:8761/eureka/,  
                  http://eureka2.tacocloud.com:8762/eureka/
```

When the service starts, it attempts to register with the first server in the zone. If that fails for any reason, then it attempts to register with the next one in the list. Eventually, when the failing Eureka comes back online, it replicates the registry from its peer, containing a registry entry for the service.

Registering a service in Eureka is only half of the story. Once services are registered in Eureka, other services can discover them and start consuming them. Let's see how to consume services registered with Eureka.

13.3.2 Consuming services

It would be a mistake to hard code any service instance's URL in the consumer's code. This not only couples the consumer to a specific instance of the service, but also can cause the consumer to break if the service's host and/or port were to change.

On the other hand, the consuming application has a lot of responsibility when it comes to looking up a service in Eureka. Eureka may reply to the lookup with many instances for the same service. If the consumer asks for ingredient-service and receives a half-dozen or so service instances in return, how does it choose the correct service?

The good news is that consuming applications don't need to make that choice or even explicitly look up a service on their own. Spring Cloud's Eureka client support, along with the Ribbon client-side load balancer, makes it simple work to look up, select, and consume a service instance. Two ways to consume a service looked up from Eureka include:

- A load-balanced RestTemplate
 - Feign-generated client interfaces

Which you choose is largely a matter of personal taste. We'll look at both options, starting with the load-balanced `RestTemplate`. Then you can choose which you like best.

CONSUMING SERVICES WITH RESTTEMPLATE

You got your first look at Spring's RestTemplate client in chapter 7. As a quick reminder of how it works, once a RestTemplate has been created or injected, you can make an HTTP call and have the responses bound to domain types. For example, to perform an HTTP GET request to retrieve an ingredient by its ID, you could use the following RestTemplate code:

The only problem with this code is that the URL passed into `getForObject()` is hard-coded to a specific host and port. I suppose that you could extract that detail into a property, but if the request is destined for one of many instances of an ingredient service, then any URL you configure would target a specific instance; there'd be no load balancer in play to spread requests across the service instances.

Once you've enabled an application as being a Eureka client, however, you have the option of declaring a load-balanced `RestTemplate` bean. All you need to do is declare a regular `RestTemplate` bean, but annotate the `@Bean` method with `@LoadBalanced`:

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

The `@LoadBalanced` annotation has two purposes: first, and most importantly, it tells Spring Cloud that this `RestTemplate` should be instrumented with the ability to look up services through Ribbon. Secondly, it acts as an injection qualifier, so that if you have two or more `RestTemplate` beans, you can specify that you want the load-balanced `RestTemplate` at the injection point.

For example, suppose that you want to use the load-balanced `RestTemplate` to look up an ingredient as in the previous code. First, you'd inject the load-balanced `RestTemplate` into the bean that needs it:

```
@Component
public class IngredientServiceClient {

    private RestTemplate rest;

    public IngredientServiceClient(@LoadBalanced RestTemplate rest) {
        this.rest = rest;
    }

    ...
}
```

Then rewrite the `getIngredientById()` method slightly so that it uses the service's registered name instead of an explicit host and port:

```
public Ingredient getIngredientById(String ingredientId) {
    return rest.getForObject(
        "http://ingredient-service/ingredients/{id}",
        Ingredient.class, ingredientId);
}
```

Did you notice the difference? The URL given to `getForObject()` doesn't use any specific hostname or port. In place of the hostname and port, the service name `ingredient-service` is used. Internally, `RestTemplate` asks Ribbon to look up a service

by that name and to select an instance. Ribbon, happy to oblige, rewrites the URL to include the host and port information for the chosen service instance and then lets `RestTemplate` proceed as usual.

As you can see, using a load-balanced `RestTemplate` isn't that different from using a standard `RestTemplate`. The key difference is that the client code only needs to deal with service names instead of explicit hostnames and ports. But what if you're using `WebClient` instead of `RestTemplate`? Can `WebClient` also be used along with Ribbon to consume services by name?

CONSUMING SERVICES WITH WEBCLIENT

In chapter 11, you saw how `WebClient` offers an HTTP client similar to `RestTemplate`, but that works with reactive types such as `Flux` and `Mono`. If you've been bit by the reactive programming bug, you might prefer to use `WebClient` instead of `RestTemplate`. The good news is that you can use `WebClient` as a load-balanced client in much the same way as you've seen `RestTemplate` used. The first thing to do is declare a `WebClient.Builder` bean method that's annotated with `@LoadBalanced`:

```
@Bean  
@LoadBalanced  
public WebClient.Builder webClientBuilder() {  
    return WebClient.builder();  
}
```

With a `WebClient.Builder` bean declared, you can now inject the load-balanced `WebClient.Builder` into any bean that needs it. For example, you might inject it into the constructor of `IngredientServiceClient`:

```
@Component  
public class IngredientServiceClient {  
  
    private WebClient.Builder wcBuilder;  
  
    public IngredientServiceClient(  
        @LoadBalanced WebClient.Builder webclientBuilder wcBuilder) {  
        this.wcBuilder = wcBuilder;  
    }  
  
    ...  
}
```

Finally, when you're ready to use it, you can use the `WebClient.Builder` to build a `WebClient` and then make requests using the service's name as it's registered in Eureka:

```
public Mono<Ingredient> getIngredientById(String ingredientId) {  
    return wcBuilder.build()  
        .get()  
        .uri("http://ingredient-service/ingredients/{id}", ingredientId)  
        .retrieve().bodyToMono(Ingredient.class);  
}
```

As with the load-balanced RestTemplate, there's no need to explicitly specify a host or port when making requests. The service name will be extracted from the given URL and used to look up a service from Eureka. Then Ribbon will select an instance of the service and the URL will be rewritten with the chosen instance's host and port before making the request.

Although this programming model is easy to grasp, especially if you're already familiar with RestTemplate or WebClient, Spring Cloud has another trick up its sleeve. Next, let's take a look at how to use Feign to create interface-based service clients.

DEFINING FEIGN CLIENT INTERFACES

Feign is a REST client library that applies a unique, interface-driven approach to defining REST clients. Put simply, if you enjoy how Spring Data automatically implements repository interfaces, then you're going to love Feign.

Feign was originally a Netflix project, but has since been turned loose as an independent open-source project called OpenFeign (<https://github.com/OpenFeign>). The word *feign* means “to pretend,” which you’ll soon see is an appropriate name for a project that pretends to be a REST client.

The first step to using Feign is to add the dependency to the project build. In pom.xml, the following <dependency> does the trick:

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

This same starter dependency can be added automatically by checking the Feign checkbox when using Spring Initializr. Unfortunately, there's no autoconfiguration to enable Feign based on the existence of this dependency. Therefore, you'll need to add the @EnableFeignClients annotation to one of the configuration classes:

```
@Configuration
@EnableFeignClients
public RestClientConfiguration {
}
```

Now comes the fun part. Let's say that you want to write a client that fetches an Ingredient from the service that's registered in Eureka as ingredient-service. The following interface is all you need:

```
package tacos.ingredientclient.feign;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import tacos.ingredientclient.Ingredient;

@FeignClient("ingredient-service")
public interface IngredientClient {
```

```

    @GetMapping("/ingredients/{id}")
    Ingredient getIngredient(@PathVariable("id") String id);

}

```

It's a simple interface, with no implementations. But at runtime, when Feign gets hold of it, none of that matters. Feign automatically creates an implementation and exposes it as a bean in the Spring application context.

Looking closer, you'll see that there are a few annotations in play that make this come together. The `@FeignClient` annotation at the interface-level specifies that any methods you declare in this interface will make requests against the service whose name is `ingredient-service`. Internally, this service will be looked up via Ribbon, the same way as it worked for the load-balanced `RestTemplate`.

Then there's the `getIngredient()` method, annotated with `@RequestMapping`. You'll no doubt recognize `@GetMapping` from Spring MVC. Indeed, it's the very same annotation! But this time, it's on the client instead of the controller. It says that any calls to `getIngredient()` will result in a GET request to the `/ingredients/{id}` path at the host and port chosen by Ribbon. The `@PathVariable` annotation, also from Spring MVC, maps the method parameter to the placeholder in the given path.

All that's left is to inject the Feign-implemented interface wherever it's needed and start using it. For instance, to use it in a controller, you might do something like this:

```

@Controller
@RequestMapping("/ingredients")
public class IngredientController {

    private IngredientClient client;

    @Autowired
    public IngredientController(IngredientClient client) {
        this.client = client;
    }

    @GetMapping("/{id}")
    public String ingredientDetailPage(@PathVariable("id") String id,
                                       Model model) {
        model.addAttribute("ingredient", client.getIngredient(id));
        return "ingredientDetail";
    }
}

```

I don't know about you, but I think that's mighty slick! It's hard to decide which I like best: the load-balanced `RestTemplate`, `WebClient`, or this magical Feign client interface. Whichever you choose, you can rest assured that your REST clients (no pun intended) will be able to consume services registered in Eureka by their name, without being hard-coded with any specific hostname or port.

For what it's worth, Feign comes with its own set of annotations. `@RequestLine` and `@Param` are roughly analogous to Spring MVC's `@RequestMapping` and `@PathVariable`,

but their use is slightly different. It's rather nice, though, to be able to use Spring MVC annotations on clients that are already familiar and, perhaps, identical to the ones you used when defining the service controllers.

Summary

- Spring Cloud Netflix enables the simple creation of a Netflix Eureka service registry with autoconfiguration and the `@EnableEurekaServer` annotation.
- Microservices register themselves by name with Eureka for discovery by other services.
- On the client-side, Ribbon acts as a client-side load balancer, looking up services by name and selecting an instance.
- Client code has the choice of either a `RestTemplate` that's instrumented for Ribbon load balancing or defining its REST client code as interfaces that are implemented automatically at runtime by Feign.
- In any event, client code is not hard-coded with the location of the services that it consumes.

Managing configuration

This chapter covers

- Running Spring Cloud Config Server
- Creating Config Server clients
- Storing sensitive configuration
- Automatically refreshing configuration

Anyone who has bought a house or a car has probably encountered a thick stack of paper. The contracts you sign when making major purchases tend to thumb their nose at the promise of a paperless society. Whenever I sit across the table from a car dealer or a title agent, I feel as if I should request a stack of bandages before getting started, in preparation for the paper cuts I'll almost certainly receive before we're done.

I've noticed that although the number of pages I must sign has stayed constant in recent years, I don't have to fill out as many fields on the forms as I once did. Where forms were once manually filled in, modern forms are more often prepopulated with basic data that was gathered before the forms were printed. This has not only made the process quicker, but has also reduced mistakes resulting from manual duplication of data across multiple forms.

Likewise, many applications have some form of configuration in play. In chapter 5 we talked about ways you can configure Spring Boot applications by setting configuration properties. Often, properties you might set are unique to the application, and it's easy enough to specify those properties in the `application.properties` or `application.yml` file that's packaged in your application's deployment.

When an application is architected with microservices, however, configuration properties are often common across multiple services. Just as it once was tedious and error-prone to manually fill in forms with duplicate data, duplicating configuration across multiple application services can be problematic.

In this chapter, we'll look at Spring Cloud's Config Server, a service that provides centralized configuration for all services in a given application. With Config Server, you can manage all of an application's configuration in one place, without duplication.

Before we get started, let's briefly consider the problems of configuring microservices individually, and how centralized configuration is better.

14.1 **Sharing configuration**

As you saw in chapter 5, you can configure Spring applications by setting properties in any of several property sources. If a configuration property is likely to change or be unique to the runtime environment, Java system properties or operating system environment variables are a fitting choice. For properties that are unlikely to change and are specific to a given application, placing those property specifications in `application.yml` or `application.properties` to be deployed with the packaged application is a fine choice.

These choices are okay for simple applications. But when you're setting configuration properties in environment variables or Java system properties, you must accept that changing those properties will require the application to be restarted. And if you choose to package the properties inside the deployed JAR or WAR file, you must completely rebuild and redeploy the application should those properties need to change. These same constraints are in play should you need to roll back changes to configuration.

Those limitations may be acceptable in some applications. In others, redeploying and restarting the application just to change a simple property is inconvenient at best and crippling at worst. Moreover, in microservice-architected applications, property management is spread across multiple codebases and deployment instances, making it unreasonable to apply the same change in every single instance of multiple services in a running application.

Some properties are sensitive, such as database passwords and other types of secrets. Although those values can be encrypted when written to an individual application's properties, the application must include the ability to decrypt those properties before they can be used. Even then, some properties may need to be kept from even the application developers, making it highly undesirable to set them in environment variables or manage them with the same source code control system as the rest of the application code.

In contrast, consider how those scenarios play out when configuration management is centralized:

- Configuration is no longer packaged and deployed with the application code, making it possible to change or roll back configuration without rebuilding or redeploying the application. Configuration can be changed on the fly without even restarting the application.
- Microservices that share common configuration needn't manage their own copy of the property settings and can share the same properties. If changes to the properties are required, those changes can be made once, in a single place, and applied to all microservices.
- Sensitive configuration details can be encrypted and maintained separate from the application code. The unencrypted values can be made available to the application on demand, rather than requiring the application to carry code that decrypts the information.

Spring Cloud Config Server provides centralized configuration with a server that all microservices within an application can rely on for their configuration. Because it's centralized, it's a one-stop shop for configuration that's common across all services, but it's also able to serve configuration that's specific to a given service.

The first step in using Config Server is to create and run the server.

14.2 Running Config Server

Spring Cloud Config Server provides a centralized source for configuration data. Much like Eureka, Config Server can be considered just another microservice whose role in the greater application is to serve configuration data for other services in the same application.

Config Server exposes a REST API through which clients (which are other services) can consume configuration properties. The configuration that's served through the Config Server is housed external to the Config Server, typically in a source code control system such as Git. Figure 14.1 illustrates how this works.

Take note of the fact that the box in figure 14.1 has the Git logo, but not the GitHub logo. That's significant—you can use any implementation of Git to store your configuration, including but not limited to GitHub, GitLab, Microsoft's Team Foundation Server, or Gogs are all valid choices as a backend for Config Server.

NOTE Although it makes little difference which Git server you use with Config Server, I'm using Gogs (<http://gogs.io>), a lightweight, easy-to-set-up Git server. More specifically, I'm running Gogs on my development machine using the instructions for running Gogs in Docker at <https://github.com/gogits/gogs/tree/master/docker>.

By storing the configuration in a source code control system such as Git, the configuration can be versioned, branched, labeled, reverted, and blamed, just like application

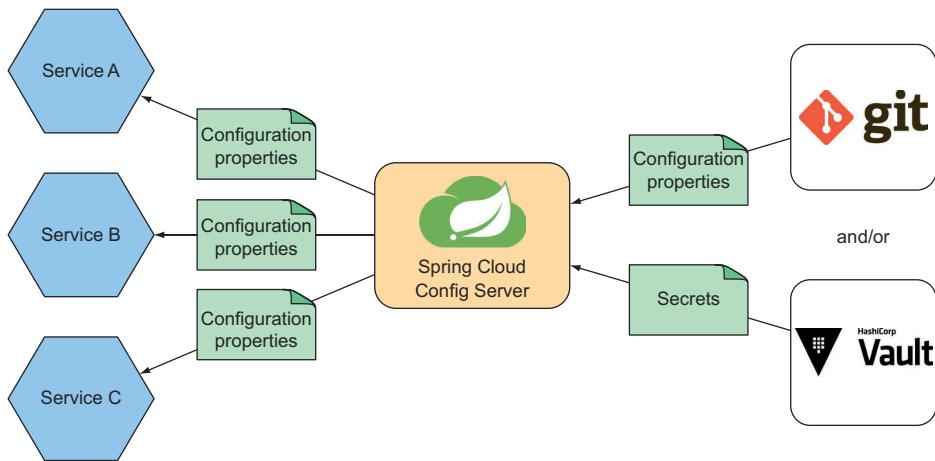


Figure 14.1 Spring Cloud Config Server serves configuration properties from a backing Git repository or Vault secret store to other services.

source code. But by keeping the configuration separate from the applications that consume it, it can evolve and be versioned independently of those applications.

You probably also noticed that HashiCorp Vault is included in figure 14.1. Vault is especially useful when the configuration properties you wish to serve are to be kept completely secret and locked away until they’re needed. We’ll talk more about using Vault with Config Server in section 14.5.

14.2.1 Enabling Config Server

As another microservice within a greater application system, Config Server is developed and deployed as a distinct application. Therefore, you’ll need to create a brand new project for Config Server. The easiest way to do this is with the Spring Initializr or one of its clients (such as the New Spring Starter Project wizard in Spring Tool Suite).

Configuration: the overloaded term

When talking about Spring Cloud Config Server, the term “configuration” gets thrown about a lot, and it isn’t always referring to the same thing. There are configuration properties that you’ll write to configure the Config Server itself. There are also configuration properties that the Config Server will serve to your applications. And the Config Server itself has the word “Config” in its name, adding slightly to the confusion.

I’ll do my best to make it clear which configuration I’m referring to whenever I use the word “configuration,” and I’ll always refer to the Config Server with the shortened word “Config.”

I’m inclined to name the new project “config-server”, but you’re welcome to name it however you wish. The most important thing to do is to specify the Config Server

dependency by checking the Config Server check box. This will result in the following dependency being added to the produced project's pom.xml file:

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

The Config Server version is ultimately determined by the Spring Cloud release train that's chosen. It's the version of the Spring Cloud release train that must be configured in the pom.xml file. At the time I'm writing this, the latest Spring Cloud release train version is Finchley.SR1. As a result, you'll also find the following property and <dependencyManagement> block in the pom.xml file:

```
<properties>
  ...
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>

...

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Although the Config Server dependency adds Spring Cloud to the project's classpath, there's no autoconfiguration to enable it, so you'll need to annotate a configuration class with @EnableConfigServer. This annotation, as its name implies, enables a Config Server when the application is running. I usually just drop @EnableConfigServer on the main class like so:

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

There's only one more thing that must be done before you can fire up the application and see the Config Server at work: you must tell it where the configuration properties that it's to serve can be found. To start, you'll use configuration that's served from a

Git repository, so you'll need to set the `spring.cloud.config.server.git.uri` property with the URL of the configuration repository:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/tacocloud/tacocloud-config
```

You'll see how to populate the Git repository with properties in section 14.2.2.

First, though, there's one other property you may wish to set for local development purposes. When testing your services locally, you'll end up having several services all running and listening on different ports on localhost. As a typical Spring Boot web application, the Config Server will listen on port 8080 by default. To avoid port collisions, you'll want to specify a unique port number by setting `server.port`:

```
server:
  port: 8888
```

Here you've set `server.port` to 8888 because, as you'll see in section 14.3, that's the default port that the configuration clients will attempt to retrieve configuration from. You're welcome to set it to any value you wish, but you'll need to be sure to configure the configuration client services to match.

It's important to realize that the configuration you've written thus far in this section is configuration for the Config Server itself. It's not the same configuration that will be served by the Config Server. Config Server will serve configuration that it pulls from Git or Vault.

At this point, if you start the application, you'll have a Config Server listening for requests on port 8888, but it will be serving absolutely no configuration properties. You don't have any Config Server clients yet, but you can pretend to be one by using the curl command-line client (or an equivalent HTTP client of your choosing):

```
$ curl localhost:8888/application/default
{
  "name": "application",
  "profiles": [
    "default"
  ],
  "label": null,
  "version": "ca791b15df07ce41d30c24937eece4ec4b208f4d",
  "state": null,
  "propertySources": []
}
```

The request made here is an HTTP GET request for the path `/application/default` on the Config Server. This path is made up of two or three parts, as illustrated in figure 14.2.

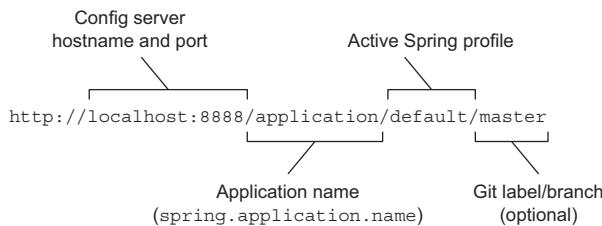


Figure 14.2 Config Server exposes a REST API through which configuration properties can be consumed.

The first part of the path, “application”, is the name of the application making the request. You’ll see later in section 14.4.1 how Config Server can use this part of the request path to serve application-specific configuration. For now you don’t have any application-specific configuration, so any value will do.

The second part in the path is the name of the Spring profile that’s active in the application making the request. In section 14.4.2 we’ll look at how Config Server can use this profile in the request path to serve configuration that’s specific to an active profile. You don’t yet have any profile-specific configuration, so any profile name will work for now.

The third part of the path, which is optional, specifies the label or branch in the backend Git repository from which to pull configuration. If not specified, this defaults to the “master” branch.

The response gives us some basic information about what the Config Server is serving, including the version and label of the Git commit that it’s serving configuration from. What are clearly missing, however, are any actual configuration properties. Normally you’d see them in the `propertySources` property, but it’s empty in this response. That’s because you still need to populate the Git repository with properties for the Config Server to serve. Let’s take care of that now.

14.2.2 Populating the configuration repository

There are several ways to set up properties for the Config Server to serve. The most basic, straightforward option is to commit an `application.properties` or `application.yml` file to the root path of the Git repository.

Let’s say you’ve pushed a file named `application.yml` to the Git repository configured in the previous section. This configuration file isn’t the same as the one you configured in the previous section; it’s the configuration that will be served by the Config Server. Suppose that in that `application.yml` file you’ve configured the following properties:

```

server:
  port: 0

eureka:
  client:
    service-url:
      defaultZone: http://eureka1:8761/eureka/
  
```

Although there isn’t much in this application.yml, what it does configure is rather significant. It tells every service in the application to choose a randomly available port and where it can register with Eureka. That means that when you adapt the services into Config Server clients in section 14.3, you’ll be able to remove explicit Eureka configuration from the services.

Acting as a client of the Config Server, you can use curl at the command line to see this new configuration data served from the Config Server:

```
$ curl localhost:8888/someapp/someconfig
{
  "name": "someapp",
  "profiles": [
    "someconfig"
  ],
  "label": null,
  "version": "95df0cbc3bca106199bd804b27a1de7c3ef5c35e",
  "state": null,
  "propertySources": [
    {
      "name": "http://localhost:10080/habuma/tacocloud-
config/application.yml",
      "source": {
        "server.port": 0,
        "eureka.client.service-url.defaultZone":
          "http://eureka1:8761/eureka/"
      }
    }
  ]
}
```

Unlike your earlier request to the Config Server, this response has stuff in the propertySources array. Specifically, it contains a property source whose name property references the Git repository and whose source contains the properties you’ve pushed into the Git repository.

SERVING CONFIGURATION FROM GIT SUBPATHS

If it suits your organizational style, you may choose to store configuration in the Git repository in a subpath instead of at the root. For example, suppose you want to put the configuration in a subdirectory named “config” relative to the root of the Git repository. If so, setting spring.cloud.config.server.git.search-paths as follows will tell the Config Server to serve configuration from /config instead of from the root:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          search-paths: config
```

Notice that the `spring.cloud.config.server.git.search-paths` property is plural. That means you can have Config Server serve from multiple paths by listing them, separated by commas:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          search-paths: config,moreConfig
```

This sets up Config Server to serve configuration from both the `/config` and `/moreConfig` paths in the Git repository.

You may also use wildcards when specifying search paths:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          search-paths: config,more*
```

Here, Config Server will serve configuration from `/config` as well as any subdirectory whose name begins with “more.”

SERVING CONFIGURATION FROM A BRANCH OR LABEL

By default, Config Server serves configuration from the master branch in Git. From the client, a specific branch or label can be specified as a third member of the request path to Config Server, as you saw in figure 14.2. But you might find it useful to have Config Server default to a specific label or branch in Git instead of the master branch. The `spring.cloud.config.server.git.default-label` property overrides the default label or branch.

For instance, consider the following configuration that sets up Config Server to serve configuration from a branch (or label) named “sidework”:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          default-label: sidework
```

As configured here, configuration will be served from the “sidework” branch unless otherwise specified by the Config Server client requesting configuration.

AUTHENTICATING WITH THE GIT BACKEND

It's quite likely that the backend Git repository your Config Server retrieves configuration from will be secured behind a username and password. In that case, you'll definitely need to provide the Config Server with the credentials for your Git repository.

The `spring.cloud.config.server.username` and `spring.cloud.config.server.password` properties set the username and password for the backend repository. The following Config Server configuration shows how you might set these properties:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          username: tacocloud
          password: s3cr3tP455w0rd
```

This sets the `username` and `password` to `tacocloud` and `s3cr3tP455w0rd`, respectively.

Using curl to pretend to be a Config Server client helps to give you some idea of how the Config Server works. And there's a lot more that Config Server can do. But the microservices you write won't be using curl to fetch configuration data. So before we look at any more ways that Config Server can be used to serve configuration data, let's shift our attention to the microservices and see how you can enable them as Config Server clients.

14.3 Consuming shared configuration

In addition to offering a centralized configuration server, Spring Cloud Config Server also provides a client library that, when included in a Spring Boot application's build, enables that application as a client of the Config Server.

The easiest way to turn any Spring Boot application into a Config Server client is to add the following dependency to the project's Maven build:

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

This same dependency is also available in the Spring Initializr as the check box labeled `Config Client`.

When the application is run, autoconfiguration will kick in to automatically register a property source that draws its properties from a Config Server. By default, it assumes that the Config Server is running on localhost and listening on port 8888. But if that's not the case, you can configure the location of the Config Server by setting the `spring.cloud.config.uri` property:

```
spring:
  cloud:
    config:
      uri: http://config.tacocloud.com:8888
```

Just to be clear, this property must be set local to the application that's to become a client of the Config Server, such as in the application.yml or application.properties file that's packaged and deployed with each microservice.

Now that you have a centralized configuration server, almost all configuration will be served from there, and each microservice won't need to carry much of its own configuration. Typically, you'll only need to set `spring.cloud.config.uri` to specify the location of the configuration server and `spring.application.name` to identify the application to the configuration server.

Which comes first: the Config Server or the Service Registry?

You're setting up your microservices to learn about the Eureka service registry from the Config Server. This is a common approach to avoid propagating service registry details across every single microservice in an application.

Alternatively, it's possible to have the Config Server register itself with Eureka and have each microservice discover the Config Server as it would any other service. If you prefer this model, you'll need to configure the Config Server as a discovery client and set the `spring.cloud.config.discovery.enabled` property to `true`. As a result, the Config Server will register itself in Eureka with the name "configserver."

The downside of this approach is that each service will need to make two calls at startup: one to Eureka to discover the Config Server, followed by one to Config Server to fetch configuration data.

When the application starts up, the property source provided by the Config Server client will make a request to the Config Server. Whatever properties it receives will be made available in the application's environment. What's more, those properties will be effectively cached; they'll be available even if the Config Server goes down. (We'll look at a few ways to refresh properties when they change in section 14.6.)

So far you've kept the configuration served by Config Server fairly simple, targeting all applications and any profile. But sometimes you'll need to enable configuration that's unique to a particular application or that should only be available when an application is running with a specific active profile. Let's take another look at Config Server and see a few more ways to use it, including serving application- and profile-specific properties.

14.4 Serving application- and profile-specific properties

As you'll recall, when a Config Server client starts up, it makes a request to the Config Server with a request path that contains both the application name as well as the name of an active profile. When serving configuration data, Config Server will consider these values and return application-specific and profile-specific configuration to the client.

From a client perspective, consuming application-specific and profile-specific configuration properties isn't much different than if you weren't using Config Server. An application's name is specified by setting the `spring.application.name` property (the same one used to identify the application to Eureka). And the active profile(s)

can be specified by setting the `spring.profiles.active` property (often as an environment variable named `SPRING_PROFILES_ACTIVE`).

Similarly, there's not much that needs to be done in the Config Server itself to serve properties that target a specific application or profile. What does matter, however, is how those properties are stored in the backing Git repository.

14.4.1 Serving application-specific properties

As we've discussed, one of the benefits of using Config Server is that you're able to share common configuration properties across all of the microservices in an application. That notwithstanding, there are often properties that are unique to one service and that need not (or should not) be shared across all services.

Along with shared configuration, Config Server is able to manage configuration properties that are targeted to a specific application. The trick is to name the configuration file the same as the application's `spring.application.name` property.

In the previous chapter, you used `spring.application.name` to give your microservices names that would be registered to Eureka. That same property is also used by a configuration client to identify itself to the Config Server, so that the Config Server can serve configuration specific to that application.

For example, in the Taco Cloud application where you've broken your application down into a handful of microservices named `ingredient-service`, `order-service`, `taco-service`, and `user-service`, you would have specified those names in each of the service applications' `spring.application.name` properties. Then you could create individual configuration YAML files in the Config Server's Git backend with filenames such as `ingredient-service.yml`, `order-service.yml`, `taco-service.yml`, and `user-service.yml`. The screenshot in figure 14.3 shows the files in the configuration repository as displayed in the Gogs web application.

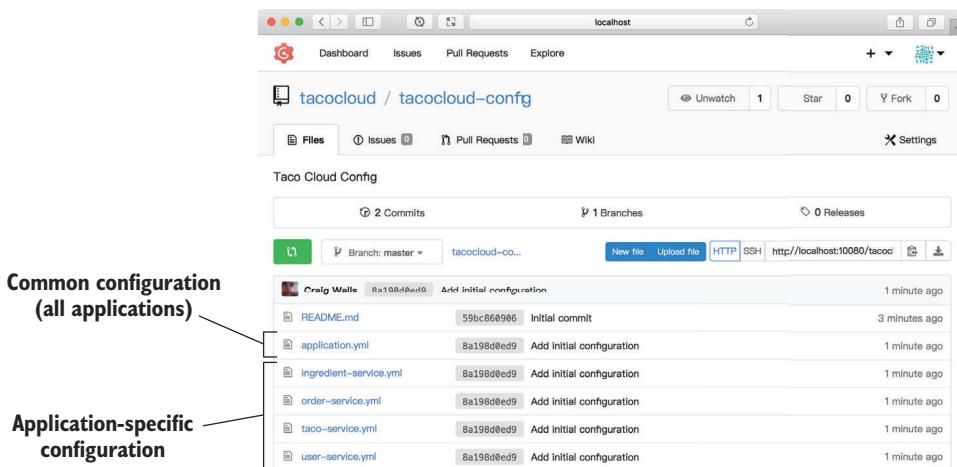


Figure 14.3 Application-specific configuration files have names based on each application's `spring.application.name` property.

No matter what an application is named, all applications will receive configuration properties from the application.yml file. But each service application's `spring.application.name` property will be sent in the request to Config Server (in the first part of the request path), and if there's a matching configuration file, those properties will also be returned. In the event of duplicate property definitions between the common properties in application.yml and those in an application-specific configuration file, the application-specific properties will take precedence.

It's worth noting that although figure 14.3 shows YAML configuration files, the same behavior holds true if properties files are checked into the Git repository.

14.4.2 Serving properties from profiles

You saw in chapter 5 how to take advantage of Spring profiles when writing configuration properties so that certain properties will only be applicable when a given profile is active. Spring Cloud Config Server supports profile-specific properties in exactly the same way that you'd use them in an individual Spring Boot application. This includes

- Providing profile-specific .properties or YAML files, such as configuration files named application-production.yml
- Including multiple profile configuration groups within a single YAML file, separated with `---` and `spring.profiles`

For example, consider the Eureka configuration that you're now sharing through the Config Server to all of your application's microservices. As it stands, it only references a single Eureka development instance. That's perfect for development environments. But if your services are running in production, you may want them to be configured with references to multiple Eureka nodes.

What's more, although you've set the `server.port` property to `0` in your development configuration, once the services go into production, they may each run in individual containers that map port `8080` to an external port, thus requiring that the applications all listen on port `8080`.

With profiles, you can declare multiple configurations. In addition to the default application.yml file that you pushed into the Config Server's Git backend, you can push another YAML file named application-production.yml that looks like this:

```
server:  
  port: 8080  
  
eureka:  
  client:  
    service-url:  
      defaultZone: http://eureka1:8761/eureka/,http://eureka2:8761/eureka/
```

When the application fetches configuration from the Config Server, it will identify which profile is active (in the second part of the request path). If the active profile is `production`, both sets of properties—`application.yml` and `application-production.yml`—will be returned, with those properties in `application-production.yml` taking precedence

over the default properties in `application.yml`. Figure 14.4 shows what this might look like in the backend Git repository.

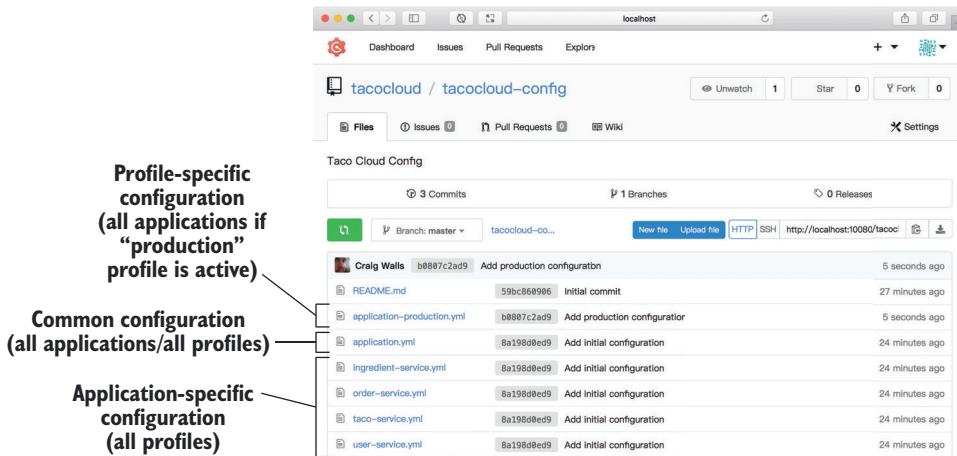


Figure 14.4 Profile-specific configuration files can be named with a suffix equal to the name of the active profile.

You can also specify properties that are specific to both a profile and an application using the same naming convention. That is, name the configuration file with the application name, hyphen, profile name.

For example, suppose you need to set properties for the application named ingredient-service that should only be applicable if the production profile is active. In that case, a configuration file named `ingredient-service-production.yml` could contain those application-specific and profile-specific properties, as illustrated in figure 14.5.

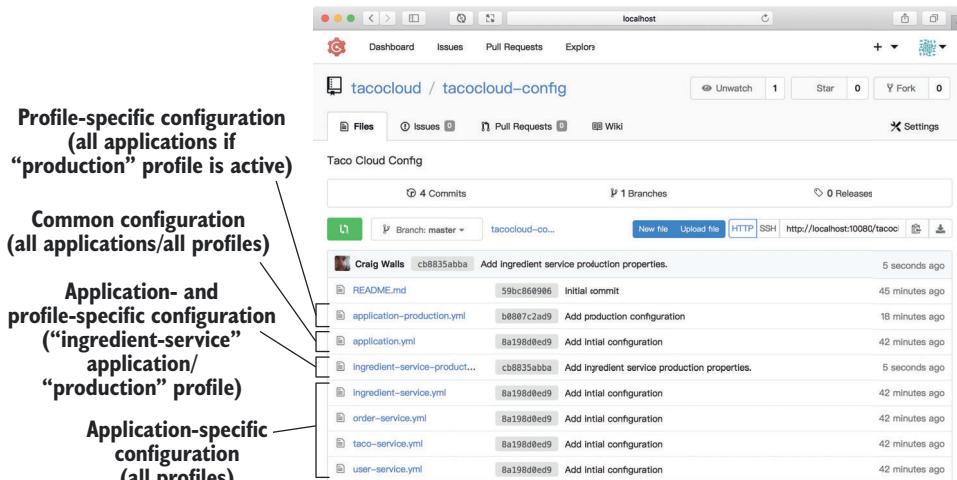


Figure 14.5 Configuration files can be both application-specific and profile-specific.

You can also use property files instead of YAML files in the backend Git repository using this same naming convention for profile-specific properties. But with YAML files, you can also include profile-specific properties in the same file as default profile properties by using a triple-hyphen separator and `spring.profiles`, as you learned in chapter 5.

14.5 Keeping configuration properties secret

Most configuration served by Config Server may not be all that secret. But you might need Config Server to serve properties containing sensitive information such as passwords or security tokens that are best kept secret in the backend repository.

Config Server offers two options for working with secret configuration properties:

- Writing encrypted values in configuration files stored in Git
- Using HashiCorp's Vault as a backend store for Config Server in addition to (or in place of) Git

Let's take a look at how each of these options can be used with Config Server to keep configuration properties secret. We'll start with writing encrypted properties to the Git backend.

14.5.1 Encrypting properties in Git

In addition to serving unencrypted values, Config Server can also serve encrypted values written in configuration files stored in Git. The key to working with encrypted data stored in Git is literally a key—an encryption key.

To enable encrypted properties, you need to configure the Config Server with an encryption key that it will use to decrypt values before serving them to its client applications. Config Server supports both symmetric and asymmetric keys. To set a symmetric key, set the `encrypt.key` property in the Config Server's own configuration to some value that will act as the encryption and decryption key:

```
encrypt:  
  key: s3cr3t
```

It's important that this property be set in bootstrap configuration (for example, `bootstrap.properties` or `bootstrap.yml`) so that it's loaded and available before autoconfiguration enables the Config Server.

For a bit tighter security, you can opt to configure Config Server with an asymmetric RSA key pair or a reference to a keystore. To create such a key, you can use the `keytool` command-line tool:

```
keytool -genkeypair -alias taccokey -keyalg RSA \  
-dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \  
-keypass s3cr3t -keystore keystore.jks -storepass 13tm31n
```

The resulting keystore will be written to a file named `keystore.jks`. You can keep the keystore file on the filesystem or place it in the application itself. In either event, you'll

need to configure the location and credentials for the keystore in the Config Server's `bootstrap.yml` file.

NOTE In order to use encryption in Config Server, you must have installed the Java Cryptography Extensions Unlimited Strength policy files. See Oracle's Java SE page for details: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

For example, suppose you choose to package the keystore in the application itself, at the root of the classpath. Then you can configure the Config Server to use that keystore with the following properties:

```
encrypt:
  key-store:
    alias: tacokey
    location: classpath:/keystore.jks
    password: 13tm3ln
    secret: s3cr3t
```

With a key or a keystore in place, you now must encrypt some data. Config Server exposes an `/encrypt` endpoint to help. All you must do is submit a POST request to the `/encrypt` endpoint with some data to be encrypted. For example, suppose you'd like to encrypt a password to the MongoDB database. Using curl, you can encrypt the password like this:

```
$ curl localhost:8888/encrypt -d "s3cr3tP455w0rd"
93912a660a7f3c04e811b5df9a3cf6e1f63850cdcd4aa092cf5a3f7e1662fab7
```

After submitting the POST request, you'll receive an encrypted value as the response. All that's left is to copy that value and paste it into the configuration that's hosted in the Git repository.

To set the MongoDB password, add the `spring.data.mongodb.password` property to the `application.yml` file stored in the Git repository:

```
spring:
  data:
    mongodb:
      password: '{cipher}93912a660a7f3c04e811b5df9a3cf6e1f63850...'
```

Notice that the value given to `spring.data.mongodb.password` is wrapped in single quotes ('') and is prefixed with `{cipher}`. This is a clue to Config Server that the value is an encrypted value and not a plain, unencrypted value.

After committing and pushing the changes in `application.yml` file to the Git repository, Config Server is ready to serve encrypted properties. To see it in action, use curl to pretend to be a Config Server client:

```
$ curl localhost:8888/application/default | jq
{
  "name": "app",
```

```

"profiles": [
    "prof"
],
"label": null,
"version": "464adfd43485182e4e0af08c2aaaa64d2f78c4cf",
"state": null,
"propertySources": [
    {
        "name": "http://localhost:10080/tacocloud/tacocloud-
config/application.yml",
        "source": {
            "spring.data.mongodb.password": "s3cr3tP455w0rd"
        }
    }
]
}

```

As you can see, the value served for `spring.data.mongodb.password` is served in a decrypted form. By default, any encrypted values served by Config Server are only encrypted while at rest in the backend Git repository; they'll be decrypted before being served. This means the client application that consumes the configuration doesn't require any special code or configuration to receive properties that are encrypted in Git.

If you'd prefer that Config Server serve encrypted properties in their still-encrypted form, you can set the `spring.cloud.config.server.encrypt.enabled` property to `false`:

```

spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          encrypt:
            enabled: false

```

This results in Config Server serving all property values, including encrypted property values, exactly as they're set in the Git repository. Pretending once more to be a client, the `curl` command reveals the effect of disabling decryption:

```

$ curl localhost:8888/application/default | jq
{
  ...
  "propertySources": [
    {
        "name": "http://localhost:10080/tacocloud/tacocloud-
config/application.yml",
        "source": {
            "spring.data.mongodb.password": "{cipher}AQA4JeVhf2cRXW..."
        }
    }
  ]
}

```

Of course, if the client is receiving encrypted property values, the client is now responsible for decrypting them on its own.

Although it's possible to store encrypted secrets in Git to be served by Config Server, we've seen that encryption is not native to Git. It requires effort on your part to encrypt any data written to the backing Git repository. Moreover, unless you push the burden of decryption to the Config Server client applications, the secret information is served decrypted through the Config Server API for anyone who may ask. Let's take a look at another Config Server backend option that only serves secrets to those who are authorized to see them.

14.5.2 Storing secrets in Vault

HashiCorp Vault is a secret-management tool. This means that in contrast to Git, Vault's core feature is handling secret information natively. For sensitive configuration data, this makes Vault a much more attractive option as a backend to Config Server.

To get started with Vault, download and install the `vault` command-line tool by following the installation instructions on the Vault website: <https://www.vaultproject.io/intro/getting-started/install.html>. In this section, you'll use the `vault` command both for managing secrets as well as starting a Vault server.

STARTING A VAULT SERVER

Before you can write secrets and serve them with Config Server, you'll need to start a Vault server. For your purposes, the easiest way to do so is to start the server in development mode with the following commands:

```
$ vault server -dev -dev-root-token-id=roottoken  
$ export VAULT_ADDR='http://127.0.0.1:8200'  
$ vault status
```

The first command starts a Vault server in development mode with a root token whose ID is `roottoken`. Development mode, as its name suggests, is a simpler, yet not entirely secure, runtime of Vault. It shouldn't be used in a production setting, but it's quite convenient when working with Vault during your development workflow.

NOTE The Vault server is a feature-filled and robust secret-management server. There's nowhere near enough space in this chapter to talk about running the Vault server beyond its simple use in development mode. I strongly recommend you get to know Vault in greater detail by reading the Vault documentation at <https://www.vaultproject.io/docs/index.html> before attempting to use Vault in a production setting.

All access to a Vault server requires that a token be presented to the server. The root token is an administrative token that, among other things, allows you to create more tokens. It can also be used to read and write secrets. If you don't specify a root token when starting the server in development mode, one will be generated for you and written to the logs at startup time. For ease of use, I recommend setting the root token to an easy-to-remember value, such as `roottoken`.

Once the development-mode server is started, it will be listening on port 8200 on the local machine. So that the `vault` command line knows where the Vault server is at, it's important to set the `VAULT_ADDR` environment variable, as in the second command in the previous code snippet.

Finally, the `vault status` command verifies that the previous two commands worked as expected. You should receive a list of about a half-dozen properties describing the configuration of the Vault server, including whether the Vault is sealed or not. (It shouldn't be sealed in development mode.)

If you're working with Vault 0.10.0 or later, there are a couple of other commands you'll need to perform to get Vault ready for working with Config Server. Some changes to how Vault works result in one of the standard secret backends being incompatible with Config Server. The following two commands recreate the backend whose name is `secret` to be compatible with Config Server:

```
$ vault secrets disable secret
$ vault secrets enable -path=secret kv
```

These steps aren't required if you're working with an older version of Vault.

WRITING SECRETS TO VAULT

The `vault` command makes it easy to write secrets into Vault. For example, suppose you want to store the password to MongoDB—the `spring.data.mongodb.password` property—in Vault instead of in Git. Using the `vault` command, you can do this:

```
$ vault write secret/application spring.data.mongodb.password=s3cr3t
```

Figure 14.6 breaks down the `vault write` command, explaining what role each part of it plays in writing the secret to Vault.

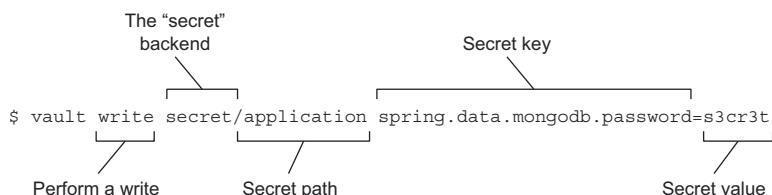


Figure 14.6 Writing a secret to Vault with the `vault` command

The most significant pieces to pay attention to for now are the secret path, key, and value. The secret path, much like a filesystem path, allows you to group related secrets in a given path and other secrets in different paths. The `secret/` prefix to the path identifies the Vault backend—in this case a key-value backend named “secret.”

The secret key and value are the actual secrets that you're writing to Vault. When writing secrets that will be served by Config Server, it's important to use secret keys that are equal to the configuration properties they'll be used for.

You can verify that the secret was written with the `vault read` command:

```
$ vault read secret/application
Key          Value
---          -----
refresh_interval    768h
spring.data.mongodb.password  s3cr3t
```

When writing secrets to a given path, be aware that every write to a given path will overwrite any secrets previously written to that path. For example, suppose you also wanted to write the MongoDB username to Vault at the same path as in the previous example. You couldn’t simply write the `spring.data.mongodb.username` secret by itself—doing so would result in the `spring.data.mongodb.password` secret being lost. Instead, you must write them both at the same time:

```
% vault write secret/application \
    spring.data.mongodb.password=s3cr3t \
    spring.data.mongodb.username=tacocloud
```

Now that you’ve written a few secrets to Vault, let’s see how you can enable Vault as a backend source of properties for Config Server.

ENABLING A VAULT BACKEND IN CONFIG SERVER

To add Vault as a backend for the Config Server, the least you’ll need to do is add `vault` as an active profile. In the Config Server’s `application.yml` file, that will look like this:

```
spring:
  profiles:
    active:
      - vault
      - git
```

As shown here, both the `vault` and `git` profiles are active, allowing Config Server to serve configuration from both Vault and Git. Generally, you’d only write sensitive configuration properties to Vault and continue to use a Git backend for those properties that don’t require secrecy. But if you wish to write all configuration to Vault or have no need for a Git backend, you can set `spring.profiles.active` to `vault` and have no Git backend at all.

By default, Config Server will assume that Vault is running on localhost, listening on port 8200. But you can change that in the Config Server’s configuration like this:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://localhost:10080/tacocloud/tacocloud-config
          order: 2
        vault:
```

```
host: vault.tacocloud.com
port: 8200
scheme: https
order: 1
```

The `spring.cloud.config.server.vault.*` properties let you override the default assumptions about Vault made by Config Server. Here you’re telling Config Server that Vault’s API can be accessed at <https://vault.tacocloud.com:8200>.

Notice that you left the Git configuration in place, assuming that Vault and Git will split responsibility for providing configuration. The `order` property specifies that secrets provided by Vault will take precedence over any properties provided by Git.

Once Config Server is configured to use the Vault backend, you can try it out by using curl to pretend to be a client:

```
[habuma:habuma] % curl localhost:8888/application/default | jq
{
  "timestamp": "2018-04-29T23:33:22.275+0000",
  "status": 400,
  "error": "Bad Request",
  "message": "Missing required header: X-Config-Token",
  "path": "/application/default"
}
```

Oh no! It looks like something went wrong! In fact, this error is an indication that Config Server is serving secrets from Vault, but that the request hasn’t included the Vault token.

It’s important that all requests to Vault include an `X-Vault-Token` header in the request. Rather than configure that token in the Config Server itself, each Config Server client will need to include the token in an `X-Config-Token` header in all requests to the Config Server. Config Server will transfer the token it receives in the `X-Config-Token` header to the `X-Vault-Token` header in requests it sends to Vault.

As you can see, for lack of a token in the request, Config Server will refuse to serve any properties, even those from Git, because Vault demands a token before divulging any of its secrets. This is an interesting side effect of using Vault alongside Git—even Git properties are indirectly hidden by the Config Server unless a valid token is provided.

Try it again, this time adding an `X-Config-Token` header to the request:

```
$ curl localhost:8888/application/default
-H "X-Config-Token: roottoken" | jq
```

The `X-Config-Token` header in the request should yield better results, including the secrets you’ve written to Vault. The token given here is the root token you specified when starting the Vault server in development mode, although it could be any token created within the Vault server that’s valid, non-expired, and that has been granted access to the Vault secret backend.

SETTING THE VAULT TOKEN IN CONFIG SERVER CLIENTS

Obviously, you won't be able to use curl in each of your microservices to specify the token when consuming properties from the Config Server. Instead, you'll need to add a little bit of configuration to each of your service application's local configuration:

```
spring:  
  cloud:  
    config:  
      token: roottoken
```

The `spring.cloud.config.token` property tells the Config Server client to include the given token value in all requests it makes to the Config Server. This property must be set in the application's local configuration (not stored in Config Server's Git or Vault backend) so that Config Server can pass it along to Vault and be able to serve properties.

WRITING APPLICATION- AND PROFILE-SPECIFIC SECRETS

When served by Config Server, secrets written to the application path will be served to all applications, regardless of their name. If you need to write secrets that are specific to a given application, replace the application portion of the path with the application name. For example, the following `vault write` command will write a secret specific to an application whose name (as identified by its `spring.application.name` property) is `ingredient-service`:

```
$ vault write secret/ingredient-service \  
  spring.data.mongodb.password=s3cr3t
```

Similarly, if you don't specify a profile, secrets written to Vault will be served as part of the default profile. That is, clients will receive those secrets regardless of what their active profile may be. You may, however, write secrets to a specific profile like this:

```
% vault write secret/application,production \  
  spring.data.mongodb.password=s3cr3t \  
  spring.data.mongodb.username=tacocloud
```

This writes the secrets such that they will only be served to applications whose active profile is `production`.

14.6 Refreshing configuration properties on the fly

As I'm writing this chapter, I'm on a plane that was pulled back to the gate for a maintenance issue. It was nothing serious, and if you're reading this, you know that the mechanics did their job satisfactorily. Even so, the interesting thing about maintenance on an airplane is that it requires the plane to be on the ground. There's not much that can be done while in flight.

In contrast, in the *Star Wars* movies, if Luke Skywalker's or Poe Dameron's X-Wing Fighter needed maintenance, the onboard mech droid could be deployed to address the issue, even while the X-Wing was in battle.

Traditionally, application maintenance, including configuration changes, has required that an application be redeployed or at least restarted. The application would need to be brought back to the gate, so to speak, for lack of a mech droid to adjust even the smallest configuration property. But that's unacceptable for cloud-native applications. We'd like to be able to change configuration properties on the fly, without even bringing the application down.

Fortunately, Spring Cloud Config Server supports the ability to refresh configuration properties of running applications with zero downtime. Once the changes have been pushed to the backing Git repository or Vault secret store, each microservice in the application can immediately be refreshed with the new configuration in one of two ways:

- *Manual*—The Config Server client enables a special Actuator endpoint at `/actuator/refresh`. An HTTP POST request to that endpoint on each service will force the config client to retrieve the latest configuration from its backends.
- *Automatic*—A commit hook in the Git repository can trigger a refresh on all services that are clients of the Config Server. This involves another Spring Cloud project called Spring Cloud Bus for communicating between the Config Server and its clients.

Each option has its pros and cons. Manual refresh gives more precise control over when services are updated with fresh configuration, but it requires an individual HTTP request to be issued to each instance of each microservice. Automatic refresh applies updated configuration instantly to all microservices in an application, but it's ultimately triggered from a commit to the configuration repository, which may be a bit too scary for some projects.

Let's take a look at each option. Then I'll let you decide which you'd prefer to apply in your projects.

14.6.1 Manually refreshing configuration properties

In chapter 16, we're going to look at the Spring Boot Actuator, one of the foundational elements of Spring Boot that enables runtime insight and some limited manipulation of runtime state, such as logging levels. But for now, we'll look at a specific Actuator feature that's only enabled in applications that are configured as Spring Cloud Config Server clients.

Whenever you enable an application to be a client of the Config Server, the auto-configuration in play also configures a special Actuator endpoint for refreshing configuration properties. To make use of this endpoint, you'll need to include the Actuator starter dependency along with the Config Client dependency in your project's build:

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

As you may have guessed, this dependency is also available from the Spring Initializr by checking the Actuator check box.

With the Actuator in play in the running config client application, you can refresh the configuration properties from the backend repositories any time you wish by submitting an HTTP POST request to /actuator/refresh.

To see this in action, let's suppose you have a @ConfigurationProperties-annotated class named GreetingProps:

```
@ConfigurationProperties(prefix="greeting")
@Component
public class GreetingProps {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Additionally, you also have a controller class that's injected with GreetingProps and that echoes the value of the message property when handling a GET request:

```
@RestController
public class GreetingController {

    private final GreetingProps props;

    public GreetingController(GreetingProps props) {
        this.props = props;
    }

    @GetMapping("/hello")
    public String message() {
        return props.getMessage();
    }
}
```

Meanwhile, in your configuration Git repository, you have an application.yml file that has the following property set:

```
greeting:
  message: Hello World!
```

With both the Config Server and this simple hello-world config client running, an HTTP GET request to /hello will yield the following response:

```
$ curl localhost:8080/hello
Hello World!
```

Now, without restarting either the Config Server or the hello-world application, change the application.yml file and push it into the backend Git repository such that the greeting.message property looks like this:

```
greeting:  
  message: Hiya folks!
```

If you make the same GET request to the hello-world application, you'll still get the same "Hello World!" response, even though the configuration has changed in Git. But you can force a refresh by POSTing to the refresh endpoint:

```
$ curl localhost:53419/actuator/refresh -X POST  
["config.client.version", "greeting.message"]
```

Notice that the response includes a JSON array of property names that have changed. Included in that array is your greeting.message property. It also includes a change to the config.client.version property, which contains the hash value of the Git commit that the current configuration comes from. Because the configuration is now based on a new Git commit, this property will change every time there's any change in the backend configuration repository.

The response from the POST request tells you that greeting.message changed. But the real proof is when you issue a GET request to the /hello path again:

```
$ curl localhost:8080/hello  
Hiya folks!
```

Without restarting the application or even restarting the Config Server, the application is now serving a brand new value for the greeting.message property!

The /actuator/refresh endpoint is great when you want full control over when an update to configuration properties takes place. But if your application is made up of several microservices (and perhaps several instances of each of those services), it can be tedious to propagate configuration to all of them. Let's take a look at how you can have configuration changes applied automatically, all at once.

14.6.2 Automatically refreshing configuration properties

As an alternative to manually refreshing properties on all Config Server clients in an application, Config Server can automatically notify all clients of changes to configuration by way of another Spring Cloud project called Spring Cloud Bus. Figure 14.7 illustrates how it works.

The property refresh process shown in figure 14.7 can be summarized like this:

- A webhook is created on the configuration Git repository to notify Config Server of any changes (such as any pushes) to the Git repository. Webhooks are supported by many Git implementations, including GitHub, GitLab, Bitbucket, and Gogs.
- Config Server reacts to webhook POST requests by broadcasting a message regarding the change by way of a message broker, such as RabbitMQ or Kafka.

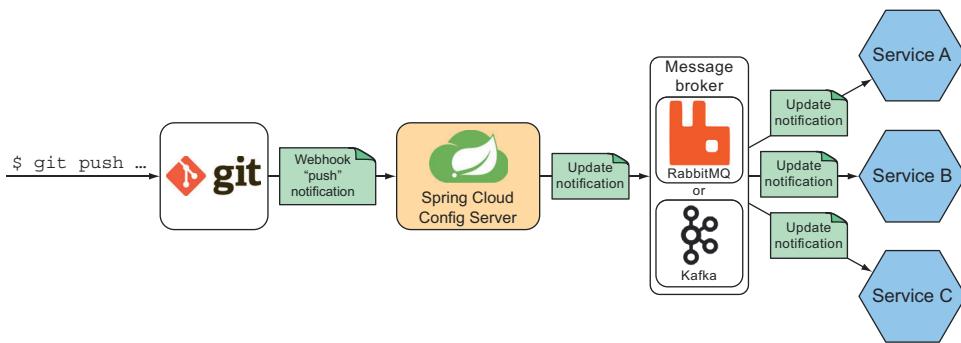


Figure 14.7 Config Server, along with Spring Cloud Bus, can broadcast changes to applications so that their properties can be refreshed automatically when there are changes.

- Individual Config Server client applications subscribed to the notifications react to the notification messages by refreshing their environments with new property values from the Config Server.

The effect is that all participating Config Server client applications will always have the latest configuration property values from Config Server almost immediately following those changes being pushed to the backend Git repository.

There are several moving parts in play when using automatic property refresh with Config Server. Let's review the changes that you're about to make to get a high-level understanding of what needs to be done:

- You'll need a message broker available to handle the messaging between Config Server and its clients. You may choose either RabbitMQ or Kafka.
- A webhook will need to be created in the backend Git repository to notify Config Server of any changes.
- Config Server will need to be enabled with the Config Server monitor dependency (which provides the endpoint that will handle webhook requests from the Git repository) and either the RabbitMQ or Kafka Spring Cloud Stream dependency (for publishing property change messages to the broker).
- Unless the message broker is running locally with the default settings, you'll need to configure the details for connecting to the broker in both the Config Server and in all of its clients.
- Each Config Server client application will need the Spring Cloud Bus dependency.

I'm going to assume that the prerequisite message broker (RabbitMQ or Kafka ... your choice) is already running and ready to channel property change messages. You'll start by applying changes to the Config Server to handle webhook update requests.

CREATING A WEBHOOK

Many Git server implementations support the creation of webhooks to notify applications of changes, including pushes, to a Git repository. The specifics of setting up

webhooks will vary from implementation to implementation, making it difficult to describe them all here. I will, however, show you how to set up a webhook for a Gogs repository.

I choose Gogs because it's easy to run locally and to have a webhook to POST to your locally running application (something that's difficult with GitHub). Also, because the process for setting up a webhook with Gogs is almost identical to that for GitHub, describing the Gogs process will indirectly give you the steps needed to set up a webhook for GitHub.

First, visit your configuration repository in the web browser and click the Settings link, as shown in figure 14.8. (The location of the Settings link is slightly different in GitHub, but it has a similar appearance.)

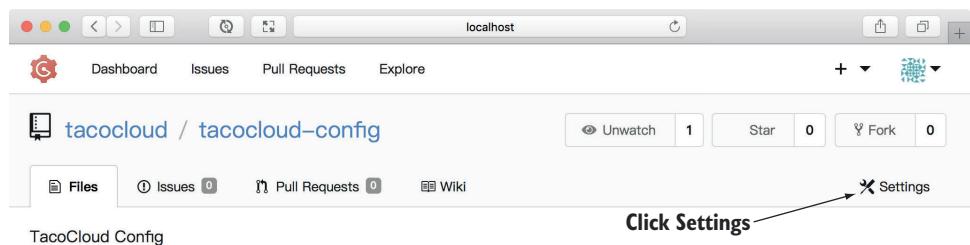


Figure 14.8 Click the Settings button in Gogs or GitHub to get started creating a webhook.

This will take you to the repository's settings page, which includes a menu of settings categories on the left. Choose the Webhooks item from the menu. This will display a page similar to what's shown in figure 14.9.

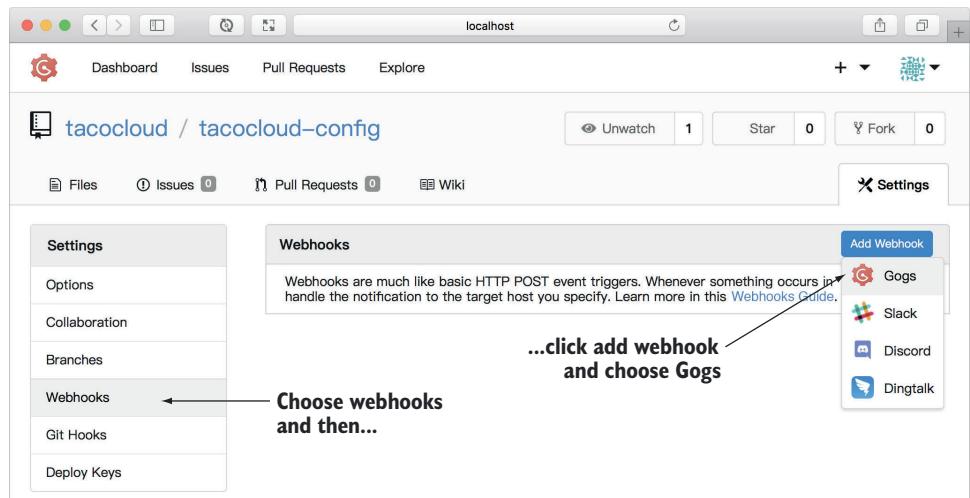


Figure 14.9 The Add Webhook button under the Webhooks menu opens the form for creating a webhook.

From the webhooks settings page, click the Add Webhook button. In Gogs, this will produce a dropdown list of options for different types of webhooks. Select the Gogs option, as shown in figure 14.9. You'll then be presented with a form to create a new webhook, as shown in figure 14.10.¹

The screenshot shows a web browser window for 'localhost' displaying the Gogs application interface. The left sidebar has a 'Settings' tab selected, with 'Webhooks' highlighted. The main content area is titled 'Add Webhook'. It contains fields for 'Payload URL' (set to 'http://host.docker.internal:8888/monitor'), 'Content Type' (set to 'application/json'), and a 'Secret' field (empty). Below these, a note states 'Secret will be sent as SHA256 HMAC hex digest of payload via X-Gogs-Signature header.' Under the heading 'When should this webhook be triggered?', there are three radio button options: 'Just the push event.' (selected), 'I need everything.', and 'Let me choose what I need.'. A checked checkbox labeled 'Active' has a note below it stating 'Details regarding the event which triggered the hook will be delivered as well.' At the bottom is a green 'Add Webhook' button.

Figure 14.10 To create a webhook, specify the Config Server's /monitor URL and JSON payload.

The Add Webhook form has several fields, but the two most significant are Payload URL and Content Type. Soon you'll be outfitting Config Server to handle webhook POST requests. When you do, Config Server will be able to handle webhook requests at a path of /monitor. Therefore, the Payload URL field should be set with a URL that

¹ GitHub doesn't have a dropdown list of webhook options. Instead, you'll be taken directly to the webhook creation form after clicking Add Webhook.

references the /monitor endpoint on your Config Server. Because I'm running Gogs in a Docker container, the URL I've given in figure 14.10 is <http://host.docker.internal:8888/monitor>, which has a hostname of host.docker.internal. This hostname enables the Gog server to see past the boundaries of its container to the Config Server running on the host machine.²

I've also set the Content Type field to application/json. This is important because the Config Server's /monitor endpoint doesn't support application/x-www-form-urlencoded, the other option for content type.

If set, the Secret field will include a header in the webhook POST request named X-Gogs-Signature (or X-Hub-Signature in the case of GitHub) that contains an HMAC-SHA256 digest (or HMAC-SHA1 for GitHub) with the given secret. At this time, Config Server's /monitor endpoint doesn't recognize the signature header, so you can leave this field blank.

Finally, you only care about push requests to the configuration repository, and you certainly wish for the webhook to be active, so you make sure that the Just the Push Event radio button and the Active check box are selected. Click the Add Webhook button at the end of the form, and the webhook will be created and will start sending POST requests to the Config Server for every push made to the repository.

Now you must enable the /monitor endpoint in Config Server to handle those requests.

HANDLING WEBHOOK UPDATES IN CONFIG SERVER

Enabling the /monitor endpoint in Config Server is a simple matter of adding the spring-cloud-config-monitor dependency to the Config Server's build. In a Maven pom.xml file, the following dependency will do the trick:

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

With that dependency in place, autoconfiguration will kick in to enable the /monitor endpoint. But it's no good unless Config Server also has a means of broadcasting the change notifications. For that, you'll need to add another dependency for Spring Cloud Stream.

Spring Cloud Stream is another one of the Spring Cloud projects; it enables the creation of services that communicate by way of some underlying binding mechanism, either RabbitMQ or Kafka. The services are written in such a way that they don't know much about how they're being used, and they accept data from the stream for processing, return data to the stream for handling by downstream services, or both.

The /monitor endpoint uses Spring Cloud Stream to publish notification messages to participating Config Server clients. To avoid being hardcoded to any particular

² In a Docker container, "localhost" means the container itself, not the Docker host.

messaging implementation, the monitor acts as a Spring Cloud Stream source, publishing messages into the stream and letting the underlying binding mechanism deal with the specifics of sending the messages.

If you’re using RabbitMQ, you’ll need to include the Spring Cloud Stream RabbitMQ binding dependency in the Config Server’s build:

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

On the other hand, if Kafka’s more your style, you’ll need the following Spring Cloud Stream Kafka dependency instead:

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

With the dependencies in place, the Config Server is almost ready to participate in automatic property refresh. In fact, if your RabbitMQ or Kafka brokers are running locally and with default settings, the Config Server is good to go. But if your message brokers are running somewhere other than `localhost`, on some nondefault port, or if you’ve changed the credentials to access the broker, you’ll need to set a few properties in the Config Server’s own configuration.

For a RabbitMQ binding, the following entries in `application.yml` can be used to override the default values:

```
spring:
  rabbitmq:
    host: rabbit.tacocloud.com
    port: 5672
    username: tacocloud
    password: s3cr3t
```

You only need to set the properties that are different for your RabbitMQ broker, even though you’ve set them all here.

If you’re using Kafka, a similar set of properties is available:

```
spring:
  kafka:
    bootstrap-servers:
      - kafka.tacocloud.com:9092
      - kafka.tacocloud.com:9093
      - kafka.tacocloud.com:9094
```

You may recognize these properties from chapter 8, where we looked at messaging with Kafka. In fact, configuring RabbitMQ and Kafka backends for automatic refresh are much the same as for any other use of those brokers in Spring.

CREATING A GOGS NOTIFICATION EXTRACTOR

Each Git implementation has its own take on what the webhook POST request should look like. This makes it important for the /monitor endpoint to be able to understand different data formats when handling webhook POST requests. Under the covers of the /monitor endpoint is a set of components that examine the POST request, try to determine what kind of Git server the request came from, and map the request data to a common notification type that's sent to each client.

Out of the box, Config Server comes with support for several popular Git implementations, such as GitHub, GitLab, and Bitbucket. If you're using one of those Git implementations, nothing special is required. But as I write this, Gogs is not yet officially supported.³ Therefore, you'll need to include a Gogs-specific notification extractor in your project if you're using Gogs as your Git implementation.

The next listing shows the notification extractor implementation that I used in Taco Cloud for Gogs integration.

Listing 14.1 A Gogs notification extractor implementation

```
package tacos.gogs;
import java.util.Collection;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import org.springframework.cloud.config.monitor.PropertyPathNotification;
import org.springframework.cloud.config.monitor.PropertyPathNotificationExtractor;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.util.MultiValueMap;

@Component
@Order(Ordered.LOWEST_PRECEDENCE - 300)
public class GogsPropertyPathNotificationExtractor
    implements PropertyPathNotificationExtractor {

    @Override
    public PropertyPathNotification extract(
        MultiValueMap<String, String> headers,
        Map<String, Object> request) {
        if ("push".equals(headers.getFirst("X-Gogs-Event"))) {
            if (request.get("commits") instanceof Collection) {
                Set<String> paths = new HashSet<>();
                @SuppressWarnings("unchecked")
                Collection<Map<String, Object>> commits =
                    (Collection<Map<String, Object>>) request
                        .get("commits");
                for (Map<String, Object> commit : commits) {
                    if (commit.get("path") != null) {
                        paths.add(commit.get("path").toString());
                    }
                }
                return new PropertyPathNotification(paths);
            }
        }
        return null;
    }
}
```

³ I submitted a pull request to the Config Server project to add Gogs support. Once it's merged, this section of the book will no longer be relevant. See <https://github.com/spring-cloud/spring-cloud-config/pull/1003> for the status of this pull request.

```

        for (Map<String, Object> commit : commits) {
            addAllPaths(paths, commit, "added");
            addAllPaths(paths, commit, "removed");
            addAllPaths(paths, commit, "modified");
        }
        if (!paths.isEmpty()) {
            return new PropertyPathNotification(
                paths.toArray(new String[0]));
        }
    }
}
return null;
}

private void addAllPaths(Set<String> paths,
                        Map<String, Object> commit,
                        String name) {
    @SuppressWarnings("unchecked")
    Collection<String> files =
        (Collection<String>) commit.get(name);
    if (files != null) {
        paths.addAll(files);
    }
}
}

```

The details of how GogsPropertyPathNotificationExtractor works is mostly irrelevant to our discussion and will become even less relevant once Spring Cloud Config Server includes Gogs support out of the box. Therefore, I won't dwell on it much and only include it here as an artifact of interest if you're using Gogs.

ENABLING AUTO-REFRESH IN CONFIG SERVER CLIENTS

Enabling automatic property refresh in Config Server clients is even easier than in Config Server itself. Only a single dependency is required:

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>

```

This adds the AMQP (for example, RabbitMQ) Spring Cloud Bus starter to the build. If you're using Kafka, you should use the following dependency instead:

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-kafka</artifactId>
</dependency>

```

With the appropriate Spring Cloud Bus starter in place, autoconfiguration will kick in as the application starts up and will automatically bind itself to a RabbitMQ broker or Kafka cluster running locally. If your RabbitMQ or Kafka is running elsewhere, you'll

need to configure its details in each client application just as you did for the Config Server itself.

Now that both the Config Server and its clients are configured for automatic refresh, fire everything up and give it a spin by making a change (any change you want) to application.yml. When you push to the Git repository, you'll immediately see the change take effect in the client applications.

Summary

- Spring Cloud Config Server offers a centralized source of configuration data to all microservices that make up a larger microservice-architected application.
- The properties served by Config Server are maintained in a backend Git or Vault repository.
- In addition to global properties, which are exposed to all Config Server clients, Config Server can also serve profile-specific and application-specific properties.
- Sensitive properties can be kept secret by encrypting them in a backend Git repository or by storing them as secrets in a Vault backend.
- Config Server clients can be refreshed with new properties either manually via an Actuator endpoint or automatically with Spring Cloud Bus and Git webhooks.

15

Handling failure and latency

This chapter covers

- Introducing the circuit breaker pattern
- Handling failure and latency with Hystrix
- Monitoring circuit breakers
- Aggregating circuit breaker metrics

15.1 Understanding circuit breakers

The circuit breaker pattern, as made popular in *Release It!, 2nd edition*, by Michael Nygard (Pragmatic Bookshelf, 2018) addresses the reality that the code we write will fail. What's important is that when it fails, it fails gracefully. This powerful pattern is even more significant in the context of microservices, where it's important to avoid letting failures cascade across a distributed call stack.

The idea of the circuit breaker pattern is relatively simple and is quite similar to a real-world electrical circuit breaker from which it gets its name. With an electrical circuit breaker, when the switch is in a closed position, the electricity flows through the circuits in a house, powering lights, televisions, computers, and appliances. But if there's any fault in the line, such as a power surge, the circuit breaker opens, stopping the flow of electricity before it damages electronics or results in a house fire.

Likewise, a software circuit breaker starts in a closed state, allowing invocations of a method. If, for any reason, that method fails (perhaps exceeding a defined threshold), the circuit opens and invocations are no longer performed against the failing method. Where a software circuit breaker differs, however, is that it provides fallback behavior and is self-correcting.

If the protected method fails within a given threshold of failure, then a fallback method can be called in its place. Once the circuit opens, that fallback method will be called almost exclusively. Every so often, though, a circuit that's open will enter a half-open state and attempt to invoke the failing method. If it still fails, the circuit resumes in an open state. If it succeeds, then it's assumed that the problem has been resolved and the circuit returns to a closed state. Figure 15.1 illustrates the flow of a software circuit breaker.

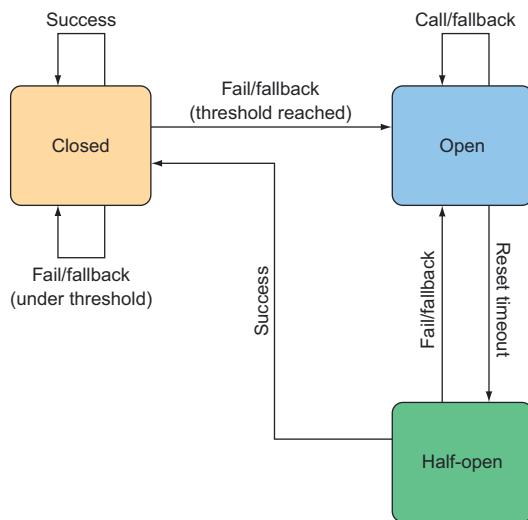


Figure 15.1 The circuit breaker pattern enables graceful failure handling.

It can be helpful to think of circuit breakers as a more powerful form of `try/catch`. A closed circuit is analogous to the `try` block, whereas the fallback method is akin to the `catch` block. Unlike `try/catch`, however, circuit breakers are intelligent enough to route calls to bypass the intended method, always calling the fallback method when the intended method is failing too frequently.

As I've implied, circuit breakers are applied on methods. There could easily be several dozen (or more) circuit breakers within a given microservice. Deciding where to declare circuit breakers in your code is a matter of identifying methods that are subject to failure. The following categories of methods are certainly candidates for circuit breakers:

- *Methods that make REST calls*—These could fail due to the remote service being unavailable or returning HTTP 500 responses.

- *Methods that perform database queries*—These could fail if, for some reason, the database becomes unresponsive, or if the schema changes in ways that break the application.
- *Methods that are potentially slow*—These won't necessarily fail, but may be considered unhealthy if they're taking too long to do their job.

That last item highlights another benefit of circuit breakers beyond handling failure. Latency is also an important concern in microservices, and it's crucial that an excessively slow method not drag down the performance of the microservice, resulting in cascading latency to upstream services.

As you can see, the circuit breaker pattern is an incredibly powerful means of gracefully handling failure and latency in code. How can we apply circuit breakers in our code? Fortunately, Netflix open source projects provide an answer with the Hystrix library.

Netflix Hystrix is a Java implementation of the circuit breaker pattern. Put simply, a Hystrix circuit breaker is implemented as an aspect applied to a method that triggers a fallback method should the target method fail. And, to properly implement the circuit breaker pattern, the aspect also tracks how frequently the target method fails and then forwards all requests to the fallback if the failure rate exceeds some threshold.

A POINT TO MAKE ABOUT HYSTRIX'S NAME

When coming up with the name for their circuit breaker implementation, the developers at Netflix wanted a name that captured the resilience, defense, and fault tolerance that would be provided. They settled on *Hystrix*, which happens to be the genus of what is known as the Old-World porcupine, an animal characterized by its ability to defend itself with long quills. Also, as explained in the Hystrix FAQ, it's a cool-sounding name. When we look at the Hystrix dashboard in section 15.3.1, you'll get to see how a porcupine found a position as the project logo.

Spring Cloud Netflix includes support for Hystrix, providing a simple programming model that should be familiar to Spring and Spring Boot developers. Declaring a circuit breaker on a method is a simple matter of annotating the method with @HystrixCommand and providing a fallback method. Let's see how to handle failure gracefully with Hystrix by declaring circuit breakers in your Taco Cloud code.

15.2 Declaring circuit breakers

Before you can declare circuit breakers, you'll need to add the Spring Cloud Netflix Hystrix starter to the build specification of each of the services. In a Maven pom.xml file, the dependency looks like this:

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

As part of the Spring Cloud portfolio, you'll also need to declare dependency management for the Spring Cloud release train in your build. As I write this, the latest release train version is Finchley.SR1. Therefore, the Spring Cloud version should be set as a property, and the following entry should appear in the pom.xml file <dependencyManagement> block:

```
<properties>
  ...
  <spring-cloud.version>Finchley.SR1</spring-cloud.version>
</properties>

...

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

NOTE This starter dependency is also available as a check box with the label Hystrix in the Initializr when creating a project. If you use the Initializr to add Hystrix to your project build, then the dependency management block is automatically created for you.

With the Hystrix starter dependency in place, the next thing you'll need to do is to enable Hystrix. The way to do that is to annotate each application's main configuration class with @EnableHystrix. For example, to enable Hystrix in the ingredient service, you'd annotate IngredientServiceApplication like this:

```
@SpringBootApplication
@EnableHystrix
public class IngredientServiceApplication {
  ...
}
```

At this point, Hystrix is enabled in your application. But that only means that all the pieces are in place for you to declare circuit breakers. You still haven't declared any circuit breakers on any of the methods. That's where the @HystrixCommand annotation comes into play.

Any method that's annotated with @HystrixCommand will be declared as having a circuit breaker aspect applied to it. For example, consider the following method that uses a load-balanced RestTemplate to fetch a collection of Ingredient objects from the ingredient service:

```
public Iterable<Ingredient> getAllIngredients() {
    ParameterizedTypeReference<List<Ingredient>> stringList =
        new ParameterizedTypeReference<List<Ingredient>>() {};
    return rest.exchange(
        "http://ingredient-service/ingredients", HttpMethod.GET,
        HttpEntity.EMPTY, stringList).getBody();
}
```

The call to `exchange()` is a potential cause for trouble. If there's no service registered in Eureka as `ingredient-service`, or if the request fails for any reason, then a `RestClientException` (an unchecked exception) will be thrown. Because the exception isn't being handled with a `try/catch` block, the caller must handle the exception. If the caller doesn't handle it, then it'll continue to be thrown upstream in the call stack. If it isn't handled at all, then the error cascades to any upstream microservices or clients.

Uncaught exceptions are a challenge in any application, but especially so in microservices. When it comes to failures, microservices should apply the Vegas Rule—what happens in a microservice, stays in a microservice. Declaring a circuit breaker on the `getAllIngredients()` method satisfies that rule.

At a minimum, you only need to annotate the method with `@HystrixCommand`, and then provide a fallback method. First, let's add `@HystrixCommand` to the `getAllIngredients()` method:

```
@HystrixCommand(fallbackMethod="getDefaultValue")
public Iterable<Ingredient> getAllIngredients() {
    ...
}
```

With a circuit breaker protecting it from failure, `getAllIngredients()` is fail safe. If, for any reason, any uncaught exceptions escape from `getAllIngredients()`, the circuit breaker will catch them and redirect the method call to a method named `getDefaultValue()`.

Fallback methods can do anything you want them to do, but the intention is that they offer backup behavior in the event that the originally intended method is unable to perform its duties. The only rule for the fallback method is that it has the same signature (aside from the method name) as the method it's serving as a backup for.

To meet this requirement, the `getAllIngredients()` method must accept no parameters and return `List<Ingredient>`. The following implementation of `getAllIngredients()` satisfies that rule and returns a default list of ingredients:

```
private Iterable<Ingredient> getDefaultValue() {
    List<Ingredient> ingredients = new ArrayList<>();
    ingredients.add(new Ingredient(
        "FLTO", "Flour Tortilla", Ingredient.Type.WRAP));
    ingredients.add(new Ingredient(
        "GRBF", "Ground Beef", Ingredient.Type.PROTEIN));
    ingredients.add(new Ingredient(
        "CHED", "Shredded Cheddar", Ingredient.Type.CHEESE));
    return ingredients;
}
```

Now, if for any reason `getAllIngredients()` fails, the circuit breaker falls back with a call to `getDefaultIngredients()`, and the caller will receive a default (albeit limited) list of ingredients.

You might be wondering if a fallback method can itself have a circuit breaker. Although there's little that could go wrong with `getDefaultIngredients()` as you've written it, it's possible that a more interesting implementation of `getDefaultIngredients()` could be a potential point of failure. In that case, you can annotate `getDefaultIngredients()` with `@HystrixCommand` and provide yet another fallback method. In fact, you can stack up as many fallback methods as make sense, if necessary. The only restriction is that there must be one method at the bottom of the fallback stack that doesn't fail and doesn't require a circuit breaker.

15.2.1 Mitigating latency

Circuit breakers can also mitigate latency by timing out if a method is taking too long to return. By default, all methods annotated with `@HystrixCommand` time out after 1 second, falling back to their declared fallback method. That means that if, for some reason, the ingredient service is sluggish in responding, then the call to `getAllIngredients()` times out after 1 second, and `getDefaultIngredients()` will be called instead.

The one-second timeout is a reasonable default and suitable for most use cases. But you can change it to be more or less restrictive by specifying a Hystrix command property. Setting Hystrix command properties can be done through the `commandProperties` attribute of the `@HystrixCommand` annotation. The `commandProperties` attribute is an array of one or more `@HystrixProperty` annotations that specify a name and a value of the property to be set.¹

In order to tweak the timeout of a circuit breaker, you need to set the Hystrix command property `execution.isolation.thread.timeoutInMilliseconds`. For example, to tighten the timeout period on the `getAllIngredients()` method to a half second, you can set the timeout to 500 as follows:

```
@HystrixCommand(
    fallbackMethod="getDefaultIngredients",
    commandProperties={
        @HystrixProperty(
            name="execution.isolation.thread.timeoutInMilliseconds",
            value="500")
    })
public Iterable<Ingredient> getAllIngredients() {
    ...
}
```

The value given is in milliseconds. If you want to loosen up the restriction, you can set it to some higher value. Or, if you don't think that there should be a timeout

¹ If you're like me, you'll agree that using annotations to set attributes of an annotation is weird. Weird or not, that's still how it's done.

imposed, then you can remove the timeout altogether by setting the command property `execution.timeout.enabled` to `false`:

```
@HystrixCommand(
    fallbackMethod="getDefaultIngredients",
    commandProperties={
        @HystrixProperty(
            name="execution.timeout.enabled",
            value="false")
    })
public Iterable<Ingredient> getAllIngredients() {
    ...
}
```

When the `execution.timeout.enabled` property is set to `false`, there's no latency protection. In this case, whether the `getAllIngredients()` method takes 1 second, 10 seconds, or 30 minutes, it won't time out. This could cause a cascading latency effect, so care should be taken when disabling execution timeouts.

15.2.2 Managing circuit breaker thresholds

By default, if a circuit breaker protected method is invoked over 20 times, and more than 50% of those invocations fail over a period of 10 seconds, the circuit will be thrown into an open state. All subsequent calls will be handled by the fallback method. After 5 seconds, the circuit will enter a half-open state, and the original method will be attempted again.

You can tweak the failure and retry thresholds by setting the Hystrix command properties. The following command properties influence the conditions that result in a circuit breaker being thrown:

- `circuitBreaker.requestVolumeThreshold`—The number of times a method should be called within a given time period
- `circuitBreaker.errorThresholdPercentage`—A percentage of failed method invocations within a given time period
- `metrics.rollingStats.timeInMilliseconds`—A rolling time period for which the request volume and error percentage are considered
- `circuitBreaker.sleepWindowInMilliseconds`—How long an open circuit remains open before entering a half-open state and the original failing method is attempted again

If both `circuitBreaker.requestVolumeThreshold` and `circuitBreaker.errorThresholdPercentage` are exceeded within the time specified in `metrics.rollingState.timeInMilliseconds`, then the circuit breaker enters an open state. It remains open for as long as specified by `circuitBreaker.sleepWindowInMilliseconds`, at which point it becomes half open, and the original failing method will be attempted again.

For example, suppose that you want to adjust the failure settings such that the method must be invoked more than 30 times and fail more than 25% of the time within 20 seconds. For that, you'll need to set the following Hystrix command properties:

```
@HystrixCommand(
    fallbackMethod="getDefaultIngredients",
    commandProperties={
        @HystrixProperty(
            name="circuitBreaker.requestVolumeThreshold",
            value="30"),
        @HystrixProperty(
            name="circuitBreaker.errorThresholdPercentage",
            value="25"),
        @HystrixProperty(
            name="metrics.rollingStats.timeInMilliseconds",
            value="20000")
    })
public List<Ingredient> getAllIngredients() {
    // ...
}
```

Additionally, should you decide that, once thrown, the circuit breaker must remain open for up to 1 full minute before becoming half open, then you can also set the `circuitBreaker.sleepWindowInMilliseconds` command property:

```
@HystrixCommand(
    fallbackMethod="getDefaultIngredients",
    commandProperties={
        ...
        @HystrixProperty(
            name="circuitBreaker.sleepWindowInMilliseconds",
            value="60000")
    })

```

Aside from gracefully handling method failures and latency, Hystrix also publishes a stream of metrics for each circuit breaker in an application. Next up, let's take a look at how to monitor the health of a Hystrix-enabled application by way of the Hystrix stream.

15.3 Monitoring failures

Every time a circuit breaker protected method is invoked, several pieces of data are collected about the invocation and published in an HTTP stream that can be used to monitor the health of the running application in real time. Among the data collected for each circuit breaker, the Hystrix stream includes the following:

- How many times the method is called
- How many times it's called successfully
- How many times the fallback method is called
- How many times the method times out

The Hystrix stream is provided by an Actuator endpoint. We'll talk more about Actuator in chapter 16. But, for now, the Actuator dependency needs to be added to the build for all the services to enable the Hystrix stream. In a Maven pom.xml file, the following starter dependency adds Actuator to a project:

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

The Hystrix stream endpoint is exposed at the path /actuator/hystrix.stream. By default, most of the Actuator endpoints are disabled. But you can enable the Hystrix stream endpoint with the following configuration in each application application.yml file like this:

```
management:
  endpoints:
    web:
      exposure:
        include: hystrix.stream
```

Optionally, the management :endpoints:web:exposure:include property can be made global for all of your services by placing it in the application.yml configuration properties that are served by the Config Server.

Application startup exposes the Hystrix stream, which can then be consumed using any REST client you want. But, before you set out to write a custom Hystrix stream client, be aware that each entry in the HTTP stream is rich with all kinds of JSON data, and it'll require a lot of client-side work to interpret that data. Although writing your own Hystrix stream presentation client isn't an impossible task, perhaps you should consider using the Hystrix dashboard before expending much effort on your own dashboard.

15.3.1 Introducing the Hystrix dashboard

To use the Hystrix dashboard, you first need to create a new Spring Boot application with a dependency on the Hystrix dashboard starter. If you're using the Spring Boot Initializr to create the project, you'll select the Hystrix Dashboard check box. Otherwise, you'll need to add the following <dependency> to your project's Maven pom.xml file:

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

Once the project has been initialized, you'll also need to enable the Hystrix dashboard by annotating the main configuration class with @EnableHystrixDashboard:

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

At development time, you'll be running the Hystrix dashboard alongside all of your other services, as well as Eureka and Config Server on your local machine. Therefore, to avoid port conflicts, you'll need to pick a unique port for the Hystrix dashboard. In the dashboard application's application.yml file, set the `server.port` property to any unique value you want. I usually set it to 7979, like this:

```
server:
  port: 7979
```

Now you're ready to fire up the Hystrix dashboard and kick the tires on it. Once it's running, open your web browser to <http://localhost:7979/hystrix>. You should see the Hystrix dashboard homepage, as shown in figure 15.2.

The first thing you'll notice about the Hystrix dashboard homepage is the logo, which is the cartoonish porcupine mascot of the Hystrix project. To start viewing a Hystrix stream, enter the URL for one of the service application Hystrix streams into

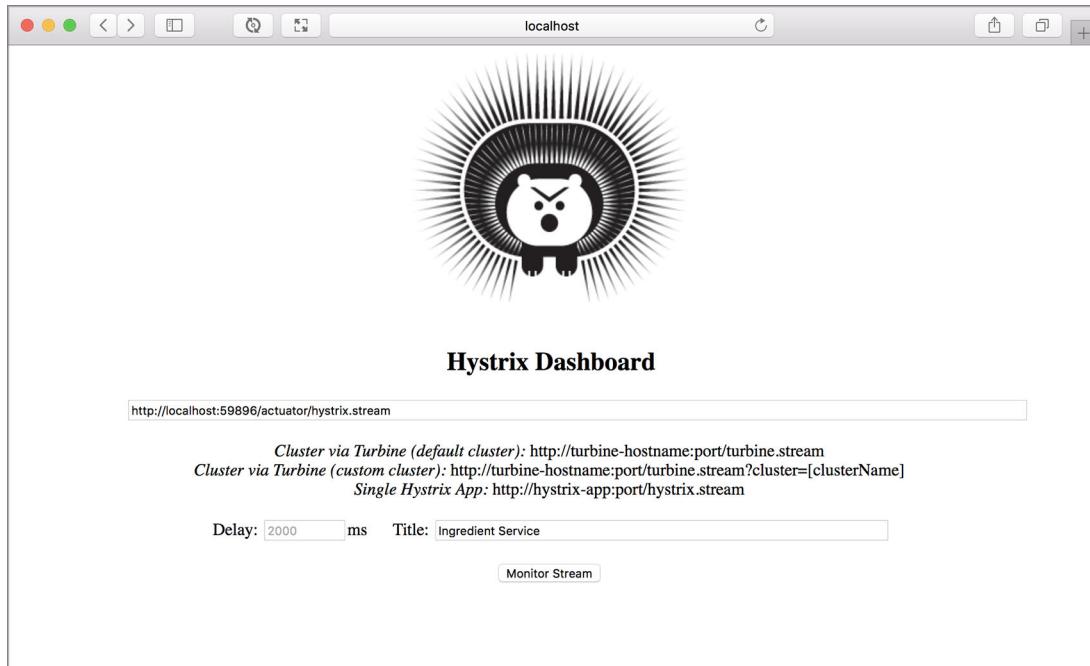


Figure 15.2 The Hystrix dashboard homepage

the text box. For example, if the ingredient service is running on localhost and listening on port 59896 (thanks to setting `server.port` to 0), then you'd enter `http://localhost:59896/actuator/hystrix.stream` into the text box.

You can also set a delay and a title to display on the Hystrix stream monitor. The delay, which defaults at 2 seconds, is the time between polling cycles, which effectively slows down the stream. The title is merely displayed as a title on the monitor page. But for your needs, the defaults are perfectly fine.

Click the Monitor Stream button to be taken to the Hystrix stream monitor. You should see a page that looks something like figure 15.3.

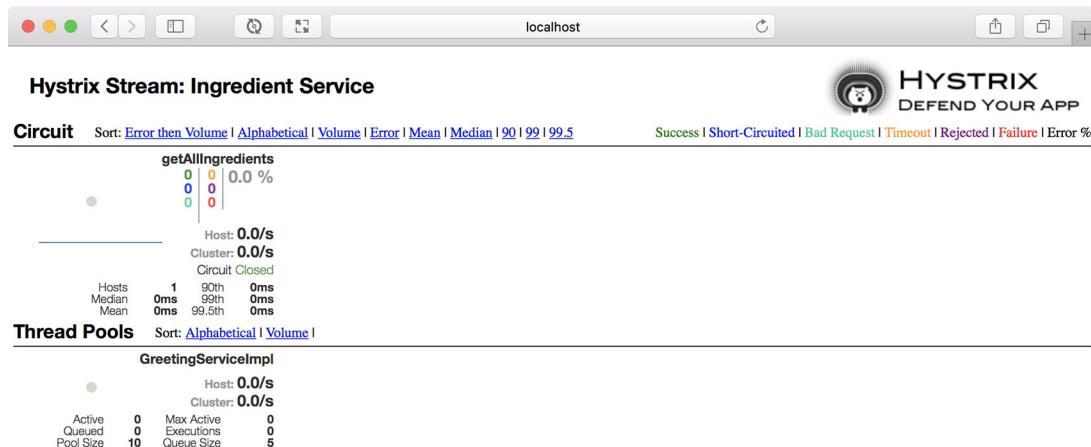


Figure 15.3 The Hystrix stream monitor page shows the metrics from each of an application's circuit breakers.

Each circuit breaker can be viewed as a graph along with some other useful metrics data. Figure 15.3 shows a single circuit breaker for `getAllIngredients()`, because that's the only circuit breaker you've declared so far.

If you don't see any graphs representing each circuit breaker and all you see is the word *Loading*, that's probably because none of the circuit breaker methods has been called yet. You must make a request to the service that would trigger a circuit breaker protected method for that method's circuit breaker metrics to appear in the dashboard. Figure 15.4 takes a closer look at an individual circuit breaker monitor, providing a breakdown of the information presented.

The most noticeable part of the monitor is the graph in the top-left corner. The line graph represents the traffic for the given method over the past 2 minutes, giving a brief history of how busy the method has been.

The background of the graph has a circle whose size and color fluctuate. The size of the circle indicates the current traffic volume; the bigger the circle grows, the higher the traffic flow. The circle color indicates its health. Green indicates healthy, yellow indicates an occasionally failing circuit breaker, and red indicates a failing circuit breaker.

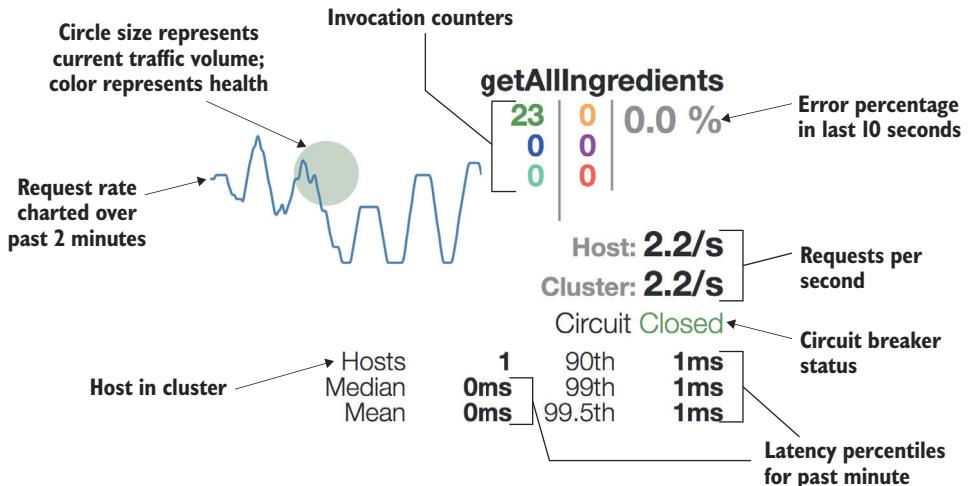


Figure 15.4 Each circuit breaker monitor provides useful information regarding the current state of the circuit breaker.

The top right of the monitor shows various counters presented in three columns. Going top-down in the leftmost column, the first number (in green—see the electronic versions of this book for color) shows how many invocations are currently successful, the second number (blue) is the number of short-circuited requests, and the last number (cyan) is the count of bad requests. The middle column shows the number of requests that have timed out (yellow), the number that the threadpool rejects (purple), and the number of failing requests (red). The third column shows a percentage of errors in the past 10 seconds.

Below the counters are two numbers representing the number of requests per second for the host and for the cluster. Below those two request rates is the status of the circuit. The bottom of the monitor shows median and mean latency, as well as the latency for the 90th, 99th, and 99.5th percentiles.

15.3.2 Understanding Hystrix thread pools

Imagine that a method is taking an excessive amount of time to do its job. Perhaps that method is making an HTTP request to another service, and the service is sluggish in responding. Until the service responds, Hystrix blocks the thread, waiting for a response.

If the method is executing in the context of the same thread as the caller of the method, then the caller doesn't have an opportunity to walk away from the long-running method. Moreover, if the blocked thread is one of a limited set of threads, such as a request-handling thread from Tomcat, and if the problem persists, then scalability can take a hit when all the threads are saturated and waiting for responses.

To avoid this situation, Hystrix assigns a thread pool for each dependency (for example, for each Spring bean with one or more Hystrix command methods). When

one of the Hystrix command methods is called, it'll be executed in a thread from the Hystrix-managed thread pool, isolating it from the calling thread. This allows the calling thread to give up and walk away from the call if it's taking too long, and isolates any potential thread saturation to the Hystrix-managed thread pool.

You may have noticed that in addition to the circuit breaker monitor, figure 15.3 also showed another monitor near the bottom of the page, under a header titled Thread Pools. This section includes a monitor for each Hystrix-managed thread pool. Figure 15.5 shows an individual thread pool monitor, annotated to describe the data it presents.

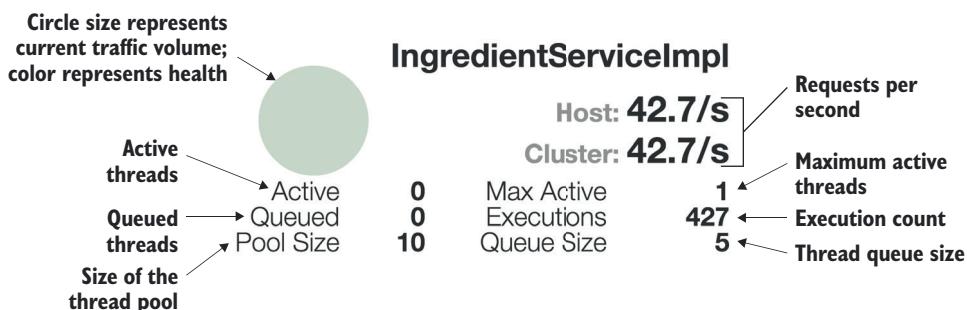


Figure 15.5 Thread pool monitors show vital statistics about each of the Hystrix-managed thread pools.

Much like the circuit breaker monitor, each thread pool monitor includes a circle in its upper-left corner. The size and color of this circle indicate how active the thread pool is currently, as well as its health. Unlike the circuit breaker monitor, however, thread pool monitors don't display a line graph showing thread pool activity over the past few minutes.

The thread pool's name is displayed in the upper-right corner, above the statistics showing the number of requests per second being handled by the threads in the thread pool. The lower-left corner of the thread pool monitor displays the following information:

- *Active thread count*—The current number of active threads.
- *Queued thread count*—How many threads are currently queued. By default, queuing is disabled, so this value is always 0.
- *Pool size*—How many threads are in the thread pool.

Meanwhile, the lower-right corner displays this information about the thread pool:

- *Maximum active thread count*—The maximum number of active threads over the current sampling period.
- *Execution count*—The number of times that threads in the thread pool have been called on to handle executions of Hystrix commands.
- *Queue size*—The size of the thread pool queue. Thread queueing is disabled by default, so this value has little meaning.

It's worth noting that as an alternative to Hystrix thread pooling, you can choose to use *semaphore isolation*. Semaphore isolation, however, is a more advanced usage of Hystrix and thus outside of the scope of this chapter. Refer to the Hystrix documentation for more information.

Now that you've seen the Hystrix dashboard in action, let's consider how to deal with multiple streams of circuit breaker data and how to aggregate them into a single stream to be viewed in the Hystrix dashboard.

15.4 Aggregating multiple Hystrix streams

The Hystrix dashboard is only capable of monitoring a single stream at a time. Because each instance of each microservice publishes its own Hystrix stream, it's almost impossible to get a holistic view of an application's health.

Fortunately, another Netflix project, Turbine, offers a way to aggregate all of the Hystrix streams from all the microservices into a single stream that the Hystrix dashboard can monitor. Spring Cloud Netflix supports creating a Turbine service using an approach similar to creating other Spring Cloud services. To create a Turbine service, create a new Spring Boot project and include the Turbine starter dependency in the build:

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

NOTE As a new project, it's easiest to simply check the Turbine check box in the Initializr when creating the new Spring Boot project.

Once you create the project, you'll need to enable Turbine. To do that, add the `@EnableTurbine` annotation to the application's main configuration class:

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

For development purposes, you'll run Turbine locally, alongside the other services in the Taco Cloud application. To avoid port conflicts, you'll need to select a unique port for Turbine so that it doesn't conflict with any of the other services. You can pick any port you like, but I tend to choose port 8989:

```
server:
  port: 8989
```

Turbine works by consuming the streams from multiple microservices and merging the circuit breaker metrics into a single stream. It acts as a client of Eureka, discovering

the services whose streams it'll aggregate into its own stream. But Turbine doesn't assume that it should aggregate the streams of all services registered in Eureka; you must configure Turbine to tell it which services it should work with.

The `turbine.app-config` property accepts a comma-delimited list of service names to look up in Eureka and for which it should aggregate Hystrix streams. For Taco Cloud, you'll need Turbine to aggregate the streams for the four services registered in Eureka as `ingredient-service`, `taco-service`, `order-service`, and `user-service`. The following entry in `application.yml` shows how to set `turbine.app-config`:

```
turbine:
  app-config: ingredient-service,taco-service,order-service,user-service
  cluster-name-expression: "default"
```

Notice that in addition to `turbine.app-config`, you also set `turbine.cluster-name-expression` to `'default'`. This indicates that Turbine should collect all of the aggregated streams under a cluster whose name is `default`. It's important to set this cluster name or else the Turbine stream won't contain any stream data aggregated from the specified applications.

Now you can fire up the Turbine server application and point your Hystrix dashboard at the stream at <http://localhost:8989/turbine.stream>. All the circuit breakers from all of the specified applications will be displayed in the circuit breaker dashboard. Figure 15.6 shows how this might look.

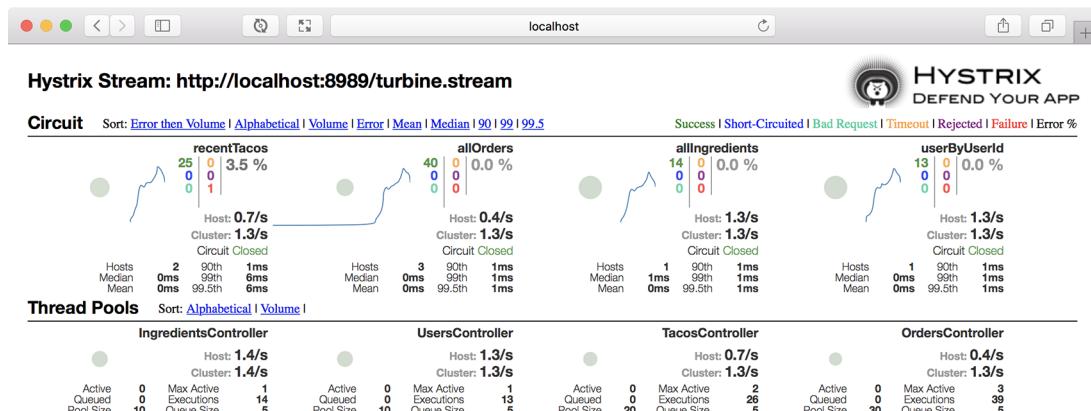


Figure 15.6 The Hystrix dashboard shows all circuit breakers from all services when pointed at an aggregated Turbine stream.

Now that the Hystrix dashboard is displaying health information for all the circuit breakers in all of your microservices—thanks to Turbine—you get a one-stop shop for monitoring the health of the circuit breakers in the Taco Cloud application.

Summary

- The circuit breaker pattern enables graceful failure handling.
- Hystrix implements the circuit breaker pattern, enabling fallback behavior when a method fails or is too slow.
- Each circuit breaker provided by Hystrix publishes metrics in a stream of data for purposes of monitoring the health of an application.
- The Hystrix stream can be consumed by the Hystrix dashboard, a web application that visualizes circuit breaker metrics.
- Turbine aggregates multiple Hystrix streams from multiple applications into a single stream that can be visualized together in the Hystrix dashboard.

Part 5

Deployed Spring

In part 5, you'll ready an application for production and see how to deploy it. Chapter 16 introduces the Spring Boot Actuator, an extension to Spring Boot that exposes the internals of a running Spring application as REST endpoints and JMX MBeans. In chapter 17 you'll see how to use the Spring Boot Admin to put a user-friendly browser-based administrative application on top of the Actuator. You'll see how to register client applications with and secure the Admin Server. Chapter 18 discusses how to expose and consume Spring beans as JMX MBeans. Finally, in chapter 19 you'll see how to deploy your Spring application in a variety of production environments. Anyone who has deployed a Java-based application may think this is obvious, but there are some features of Spring Boot and related Spring projects that make deploying Spring boot applications unique.

Working with Spring Boot Actuator

This chapter covers

- Enabling Actuator in Spring Boot projects
- Exploring Actuator endpoints
- Customizing Actuator
- Securing Actuator

Have you ever tried to guess what's inside a wrapped gift? You shake it, weigh it, and measure it. And you might even have a solid idea as to what's inside. But until you open it up, there's no way of knowing for sure.

A running application is kind of like a wrapped gift. You can poke at it and make reasonable guesses as to what's going on under the covers. But how can you know for sure? If only there were some way that you could peek inside a running application, see how it's behaving, check on its health, and maybe even trigger operations that influence how it runs!

In this chapter, we're going to explore Spring Boot's Actuator. Actuator offers production-ready features such as monitoring and metrics to Spring Boot applications. Actuator's features are provided by way of several endpoints, which are made available over HTTP as well as through JMX MBeans. This chapter focuses primarily on HTTP endpoints, saving JMX endpoints for chapter 19.

16.1 Introducing Actuator

In a machine, an actuator is a component that's responsible for controlling and moving a mechanism. In a Spring Boot application, the Spring Boot Actuator plays that same role, enabling us to see inside of a running application and, to some degree, control how the application behaves.

Using endpoints exposed by Actuator, we can ask things about the internal state of a running Spring Boot application:

- What configuration properties are available in the application environment?
- What are the logging levels of various packages in the application?
- How much memory is being consumed by the application?
- How many times has a given HTTP endpoint been requested?
- What is the health of the application and any external services it coordinates with?

To enable Actuator in a Spring Boot application, you simply need to add Actuator's starter dependency to your build. In any Spring Boot application Maven pom.xml file, the following <dependency> entry does the trick:

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

Once the Actuator starter is part of the project build, the application will be equipped with several out-of-the-box Actuator endpoints, including those described in table 16.1.

Table 16.1 Actuator endpoints for peeking inside and manipulating the state of a running Spring Boot application

HTTP method	Path	Description	Enabled by default?
GET	/auditevents	Produces a report of any audit events that have been fired.	No
GET	/beans	Describes all the beans in the Spring application context.	No
GET	/conditions	Produces a report of autoconfiguration conditions that either passed or failed, leading to the beans created in the application context.	No
GET	/configprops	Describes all configuration properties along with the current values.	No
GET, POST, DELETE	/env	Produces a report of all property sources and their properties available to the Spring application.	No
GET	/env/{toMatch}	Describes the value of a single environment property.	No
GET	/health	Returns the aggregate health of the application and (possibly) the health of external dependent applications.	Yes

Table 16.1 Actuator endpoints for peeking inside and manipulating the state of a running Spring Boot application (*continued*)

HTTP method	Path	Description	Enabled by default?
GET	/heapdump	Downloads a heap dump.	No
GET	/httptrace	Produces a trace of the most recent 100 requests.	No
GET	/info	Returns any developer-defined information about the application.	Yes
GET	/loggers	Produces a list of packages in the application along with their configured and effective logging levels.	No
GET, POST	/loggers/{name}	Returns the configured and effective logging level of a given logger. The effective logging level can be set with a POST request.	No
GET	/mappings	Produces a report of all HTTP mappings and their corresponding handler methods.	No
GET	/metrics	Returns a list of all metrics categories.	No
GET	/metrics/{name}	Returns a multidimensional set of values for a given metric.	No
GET	/scheduledtasks	Lists all scheduled tasks.	No
GET	/threaddump	Returns a report of all application threads.	No

In addition to HTTP-based endpoints, all of the Actuator endpoints in table 16.1, with the lone exception of /heapdump, are also exposed as JMX MBeans. We'll look at the JMX side of Actuator in chapter 19.

16.1.1 Configuring Actuator's base path

By default, the paths for all the endpoints shown in table 16.1 are prefixed with /actuator. This means that, for example, if you wish to retrieve health information about your application from Actuator, then issuing a GET request for /actuator/health will return the information you need.

The Actuator prefix path can be changed by setting the `management.endpoint.web.base-path` property. For example, if you'd rather the prefix be /management, you would set the `management.endpoints.web.base-path` property like this:

```
management:
  endpoints:
    web:
      base-path: /management
```

With this property set as shown, you'd need to make a GET request for /management/health to obtain the application's health information.

16.1.2 Enabling and disabling Actuator endpoints

You may have noticed that only the /health and /info endpoints are enabled by default. Most Actuator endpoints carry sensitive information and should be secured. You can use Spring Security to lock down Actuator, but because Actuator isn't secured on its own, most of the endpoints are disabled by default, requiring you to opt in for the endpoints you wish to expose.

Two configuration properties, management.endpoints.web.exposure.include and management.endpoints.web.exposure.exclude, can be used to control which endpoints are exposed. Using management.endpoints.web.exposure.include, you can specify which endpoints you want to expose. For example, if you wish to expose only the /health, /info, /beans, and /conditions endpoints, you can specify that with the following configuration:

```
management:  
  endpoints:  
    web:  
      exposure:  
        include: health,info,beans,conditions
```

The management.endpoints.web.exposure.include property also accepts an asterisk (*) as a wildcard to indicate that all Actuator endpoints should be exposed:

```
management:  
  endpoints:  
    web:  
      exposure:  
        include: '*'
```

If you want to expose all but a few endpoints, it's typically easier to include them all with a wildcard and then explicitly exclude a few. For example, to expose all Actuator endpoints except for /threaddump and /heapdump, you could set both the management.endpoints.web.exposure.include and management.endpoints.web.exposure.exclude properties like this:

```
management:  
  endpoints:  
    web:  
      exposure:  
        include: '*'  
        exclude: threaddump,heapdump
```

Should you decide to expose more than /health and /info, it's probably a good idea to configure Spring Security to restrict access to the other endpoints. We'll look at how to secure Actuator endpoints in section 16.4. For now, though, let's look at how you can consume the HTTP endpoints exposed by Actuator.

16.2 Consuming Actuator endpoints

Actuator can bestow a veritable treasure trove of interesting and useful information about a running application by way of the HTTP endpoints listed in table 16.1. As HTTP endpoints, these can be consumed like any REST API, using whatever HTTP client you wish, including Spring’s `RestTemplate` and `WebClient`, from a browser-based JavaScript application, or simply with the `curl` command-line client.

For the sake of exploring Actuator endpoints, you’ll use the `curl` command-line client in this chapter. In chapter 17, I’ll introduce you to Spring Boot Admin, which layers a user-friendly web application on top of an application’s Actuator endpoints.

To get some idea of what endpoints Actuator has to offer, a GET request to Actuator’s base path will provide HATEOAS links for each of the endpoints. Using `curl` to make a request to `/actuator`, you might get a response something like this (abridged to save space):

```
$ curl localhost:8081/actuator
{
  "_links": {
    "self": {
      "href": "http://localhost:8081/actuator",
      "templated": false
    },
    "auditevents": {
      "href": "http://localhost:8081/actuator/auditevents",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8081/actuator/beans",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8081/actuator/health",
      "templated": false
    },
    ...
  }
}
```

Because different libraries may contribute additional Actuator endpoints of their own, and because some endpoints may be not be exported, the actual results may vary from application to application.

In any event, the set of links returned from Actuator’s base path serve as a map to all that Actuator has to offer. Let’s begin our exploration of the Actuator landscape with the two endpoints that provide essential information about an application: the `/health` and `/info` endpoints.

16.2.1 Fetching essential application information

At the beginning of a typical visit to the doctor, we're usually asked two very basic questions: who are you and how do you feel? Although the words chosen by the doctor or nurse may be different, they ultimately want to know a little bit about the person they're treating and why you're seeing them.

Those same essential questions are what Actuator's /info and /health endpoints answer for a Spring Boot application. The /info endpoint tells you a little about the application, and the /health endpoint tells you how healthy the application is.

ASKING FOR INFORMATION ABOUT AN APPLICATION

To learn a little bit of information about a running Spring Boot application, you can ask the /info endpoint. By default, however, the /info endpoint isn't very informative. Here's what you might see when you make a request for it using curl:

```
$ curl localhost:8081/actuator/info
{}
```

While it may seem that the /info endpoint isn't very useful, it's best to think of it as a clean canvas on which you may paint any information you'd like to present.

There are several ways to supply information for the /info endpoint to return, but the most straightforward way is to create one or more configuration properties where the property name is prefixed with `info..`. For example, suppose that you want the response from the /info endpoint to include support contact information, including an email address and phone number. To do that, you can configure the following properties in the application.yml file:

```
info:
  contact:
    email: support@tacocloud.com
    phone: 822-625-6831
```

Neither the `info.contact.email` property nor the `info.contact.phone` property have any special meaning to Spring Boot or any bean that may be in the application context. However, by virtue of the fact that it's prefixed with `info..`, the /info endpoint will now echo the value of the property in its response:

```
{
  "contact": {
    "email": "support@tacocloud.com",
    "phone": "822-625-6831"
  }
}
```

In section 16.3.1, we'll look at a few other ways to populate the /info endpoint with useful information about an application.

INSPECTING APPLICATION HEALTH

Issuing an HTTP GET request for the /health endpoint results in a simple JSON response with the health status of your application. For example, here's what you might see when using curl to fetch the /health endpoint:

```
$ curl localhost:8080/actuator/health
{"status": "UP"}
```

You may be wondering how useful it is to have an endpoint that reports that the application is UP. What would it report if the application were down?

As it turns out, the status shown here is an aggregate status of one or more health indicators. Health indicators report the health of external systems that the application interacts with, such as databases, message brokers, and even Spring Cloud components such as Eureka and the Config Server. The health status of each indicator could be one of the following:

- UP—The external system is up and is reachable.
- DOWN—The external system is down or unreachable.
- UNKNOWN—The status of the external system is unclear.
- OUT_OF_SERVICE—The external system is reachable but is currently unavailable.

The health statuses of all health indicators are then aggregated into the application's overall health status, applying the following rules:

- If all health indicators are UP, then the application health status is UP.
- If one or more health indicators are DOWN, then the application health status is DOWN.
- If one or more health indicators are OUT_OF_SERVICE, then the application health status is OUT_OF_SERVICE.
- UNKNOWN health statuses are ignored and aren't rolled into the application's aggregate health.

By default, only the aggregate status is returned in response to a request for /health. You can configure the management.endpoint.health.show-details property, however, to show the full details of all health indicators:

```
management:
  endpoint:
    health:
      show-details: always
```

The management.endpoint.health.show-details property defaults to never, but it can also be set to always to always show the full details of all health indicators, or to when-authorized to only show the full details when the requesting client is fully authorized.

Now when you issue a GET request to the /health endpoint, you get full health indicator details. Here's a sample of what that might look like for a service that integrates with the Mongo document database:

```
{  
    "status": "UP",  
    "details": {  
        "mongo": {  
            "status": "UP",  
            "details": {  
                "version": "3.2.2"  
            }  
        },  
        "diskSpace": {  
            "status": "UP",  
            "details": {  
                "total": 499963170816,  
                "free": 177284784128,  
                "threshold": 10485760  
            }  
        }  
    }  
}
```

All applications, regardless of any other external dependencies, will have a health indicator for the filesystem named diskSpace. The diskSpace health indicator indicates the health of the filesystem (hopefully, UP), which is determined by how much free space is remaining. If the available disk space drops below the threshold, it will report a status of DOWN.

In the preceding example, there's also a mongo health indicator, which reports the status of the Mongo database. Details shown include the Mongo database version.

Autoconfiguration ensures that only health indicators that are pertinent to an application will appear in the response from the /health endpoint. In addition to the mongo and diskSpace health indicators, Spring Boot also provides health indicators for several other external databases and systems, including these:

- Cassandra
- Config Server
- Couchbase
- Eureka
- Hystrix
- JDBC data sources
- Elasticsearch
- InfluxDB
- JMS message brokers
- LDAP
- Email servers

- Neo4j
- Rabbit message brokers
- Redis
- Solr

Additionally, third-party libraries may contribute their own health indicators. We'll look at how to write a custom health indicator in section 16.3.2.

As you've seen, the /health and /info endpoints provide general information about the running application. Meanwhile, there are other Actuator endpoints that provide insight into the application configuration. Let's look at how Actuator can show how an application is configured.

16.2.2 Viewing configuration details

Beyond receiving general information about an application, it can be enlightening to understand how an application is configured. What beans are in the application context? What autoconfiguration conditions passed or failed? What environment properties are available to the application? How are HTTP requests mapped to controllers? What logging level are one or more packages or classes set to?

These questions are answered by Actuator's /beans, /conditions, /env, /configprops, /mappings, and /loggers endpoints. And in some cases, such as /env and /loggers, you can even adjust the configuration of a running application on the fly. We'll look at how each of these endpoints gives insight into the configuration of a running application, starting with the /beans endpoint.

GETTING A BEAN WIRING REPORT

The most essential endpoint for exploring the Spring application context is the /beans endpoint. This endpoint returns a JSON document describing every single bean in the application context, its Java type, and any of the other beans it's injected with.

A complete response from a GET request to /beans could easily fill this entire chapter. Instead of examining the complete response from /beans, let's consider the following snippet, which focuses on a single bean entry:

```
{  
    "contexts": {  
        "application-1": {  
            "beans": {  
                ...  
                "ingredientsController": {  
                    "aliases": [],  
                    "scope": "singleton",  
                    "type": "tacos.ingredients.IngredientsController",  
                    "resource": "file [/Users/habuma/Documents/Workspaces/  
                        TacoCloud/ingredient-service/target/classes/tacos/  
                        ingredients/IngredientsController.class]",  
                    "dependencies": [  
                        "ingredientRepository"  
                }  
            }  
        }  
    }  
}
```

```

        ],
    },
...
},
"parentId": null
}
}
}
}
```

At the root of the response is the contexts element, which includes one sub-element for each Spring application context in the application. Within each application context is a beans element that holds details for all the beans in the application context.

In the preceding example, the bean shown is the one whose name is ingredientsController. You can see that it has no aliases, is scoped as a singleton, and is of type tacos.ingredients.IngredientsController. Moreover, the resource property gives the path to the class file that defines the bean. And the dependencies property lists all other beans that are injected into the given bean. In this case, the ingredientsController bean is injected with a bean whose name is ingredientRepository.

EXPLAINING AUTOCONFIGURATION

As you've seen, autoconfiguration is one of the most powerful things that Spring Boot offers. Sometimes, however, you may wonder why something has been auto-configured. Or you may expect something to have been autoconfigured and are left wondering why it hasn't been. In that case, you can make a GET request to /conditions to get an explanation of what took place in autoconfiguration.

The autoconfiguration report returned from /conditions is divided into three parts: positive matches (conditional configuration that passed), negative matches (conditional configuration that failed), and unconditional classes. The following snippet from the response to a request to /conditions shows an example of each section:

```
{
  "contexts": {
    "application-1": {
      "positiveMatches": {
...
        "MongoDataAutoConfiguration#mongoTemplate": [
          {
            "condition": "OnBeanCondition",
            "message": "@ConditionalOnMissingBean (types:
              org.springframework.data.mongodb.core.MongoTemplate;
              SearchStrategy: all) did not find any beans"
          }
        ],
...
      },
      "negativeMatches": {
...
        "DispatcherServletAutoConfiguration": {
          "notMatched": [

```

```
        {
            "condition": "OnClassCondition",
            "message": "@ConditionalOnClass did not find required
                        class 'org.springframework.web.servlet.
                                         DispatcherServlet'"
        }
    ],
    "matched": []
},
...
},
"unconditionalClasses": [
    ...
        "org.springframework.boot.autoconfigure.context.
                                         ConfigurationPropertiesAutoConfiguration",
    ...
]
}
}
```

Under the positiveMatches section, you see that a MongoTemplate bean was configured by autoconfiguration because one didn't already exist. The autoconfiguration that caused this includes a @ConditionalOnMissingBean annotation, which passes off the bean to be configured if it hasn't already been explicitly configured. But in this case, no beans of type MongoTemplate were found, so autoconfiguration stepped in and configured one.

Under negativeMatches, Spring Boot autoconfiguration considered configuring a DispatcherServlet. But the @ConditionalOnClass conditional annotation failed because DispatcherServlet couldn't be found.

Finally, a `ConfigurationPropertiesAutoConfiguration` bean was configured unconditionally, as seen under the `unconditionalClasses` section. Configuration properties are foundational to how Spring Boot operates, so any configuration pertaining to configuration properties should be autoconfigured without any conditions.

INSPECTING THE ENVIRONMENT AND CONFIGURATION PROPERTIES

In addition to knowing how your application beans are wired together, you might also be interested in learning what environment properties are available and what configuration properties were injected on the beans.

When you issue a GET request to the /env endpoint, you'll receive a rather lengthy response that includes properties from all property sources in play in the Spring application. This includes properties from environment variables, JVM system properties, application.properties and application.yml files, and even the Spring Cloud Config Server (if the application is a client of the Config Server).

The following listing shows a greatly abridged example of the kind of response you might get from the /env endpoint, to give you some idea of the kind of information it provides.

Listing 16.1 The results from the /env endpoint

```
$ curl localhost:8081/actuator/env
{
  "activeProfiles": [
    "development"
  ],
  "propertySources": [
    ...
    {
      "name": "systemEnvironment",
      "properties": {
        "PATH": {
          "value": "/usr/bin:/bin:/usr/sbin:/sbin",
          "origin": "System Environment Property \"PATH\""
        },
        ...
        "HOME": {
          "value": "/Users/habuma",
          "origin": "System Environment Property \"HOME\""
        }
      }
    },
    {
      "name": "applicationConfig: [classpath:/application.yml]",
      "properties": {
        "spring.application.name": {
          "value": "ingredient-service",
          "origin": "class path resource [application.yml]:3:11"
        },
        "server.port": {
          "value": 8081,
          "origin": "class path resource [application.yml]:9:9"
        },
        ...
      }
    },
    ...
  ]
}
```

Although the full response from /env provides even more information, what's shown in listing 16.1 contains a few noteworthy elements. First, notice that near the top of the response is a field named `activeProfiles`. In this case, it indicates that the `development` profile is active. If any other profiles were active, those would be listed as well.

Next, the `propertySources` field is an array containing an entry for every property source in the Spring application environment. In listing 16.1, only the `systemEnvironment` and an `applicationConfig` property source referencing the `application.yml` file are shown.

Within each property source is a listing of all properties provided by that source, paired with their values. In the case of the application.yml property source, the origin field for each property tells exactly where the property is set, including the line and column within application.yml.

The /env endpoint can also be used to fetch a specific property when that property's name is given as the second element of the path. For example, to examine the server.port property, submit a GET request for /env/server.port, as shown here:

```
$ curl localhost:8081/actuator/env/server.port
{
  "property": {
    "source": "systemEnvironment", "value": "8081"
  },
  "activeProfiles": [ "development" ],
  "propertySources": [
    { "name": "server.ports" },
    { "name": "mongo.ports" },
    { "name": "systemProperties" },
    { "name": "systemEnvironment",
      "property": {
        "value": "8081",
        "origin": "System Environment Property \"SERVER_PORT\""
      }
    },
    { "name": "random" },
    { "name": "applicationConfig: [classpath:/application.yml]",
      "property": {
        "value": 0,
        "origin": "class path resource [application.yml]:9:9"
      }
    },
    { "name": "springCloudClientHostInfo" },
    { "name": "refresh" },
    { "name": "defaultProperties" },
    { "name": "Management Server" }
  ]
}
```

As you can see, all property sources are still represented, but only those that set the specified property will contain any additional information. In this case, both the systemEnvironment property source and the application.yml property source had values for the server.port property. Because the systemEnvironment property source takes precedence over any of the property sources listed below it, its value of 8081 wins. The winning value is reflected near the top under the property field.

The /env endpoint can be used for more than just reading property values. By submitting a POST request to the /env endpoint, along with a JSON document with a name and value field, you can also set properties in the running application. For example, to set a property named tacocloud.discount.code to TACOS1234, you can use curl to submit the POST request at the command line like this:

```
$ curl localhost:8081/actuator/env \
-d'{"name":"tacocloud.discount.code","value":"TACOS1234"}' \
-H "Content-type: application/json"
{"tacocloud.discount.code":"TACOS1234"}
```

After submitting the property, the newly set property and its value are returned in the response. Later, should you decide you no longer need that property, you can submit a DELETE request to the /env endpoint to delete all properties created through that endpoint:

```
$ curl localhost:8081/actuator/env -X DELETE
{"tacocloud.discount.code":"TACOS1234"}
```

As useful as setting properties through Actuator's API can be, it's important to be aware that any properties set with a POST request to the /env endpoint only apply to the application instance receiving the request, are temporary, and will be lost when the application restarts.

NAVIGATING HTTP REQUEST MAPPINGS

Although Spring MVC's (and Spring WebFlux's) programming model makes it easy to handle HTTP requests by simply annotating methods with request-mapping annotations, it can sometimes be challenging to get a big-picture understanding of all the kinds of HTTP requests that an application can handle, and what kinds of components handle those requests.

Actuator's /mappings endpoint offers a one-stop view of every HTTP request handler in an application, whether it be from a Spring MVC controller or one of Actuator's own endpoints. To get a complete list of all the endpoints in a Spring Boot application, make a GET request to the /mappings endpoint, and you might receive something that's a little bit like the abridged response shown next.

Listing 16.2 HTTP mappings as shown by the /mappings endpoint

```
$ curl localhost:8081/actuator/mappings | jq
{
  "contexts": {
    "application-1": {
      "mappings": {
        "dispatcherHandlers": [
          "webHandler": [
            ...
            {
              "predicate": "{ [/ingredients] ,methods=[GET] }",
              "handler": "public
reactor.core.publisher.Flux<tacos.ingredients.Ingredient>
tacos.ingredients.IngredientsController.allIngredients()",
              "details": {
                "handlerMethod": {
                  "className": "tacos.ingredients.IngredientsController",
                  "name": "allIngredients",
                  "signature": "(none)"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

```

        "descriptor": "()&reactor/core/publisher/Flux;"  

    },  

    "handlerFunction": null,  

    "requestMappingConditions": {  

        "consumes": [],  

        "headers": [],  

        "methods": [  

            "GET"  

        ],  

        "params": [],  

        "patterns": [  

            "/ingredients"  

        ],  

        "produces": []  

    }  

},  

...  

]  

}  

},  

"parentId": "application-1"  

},  

"bootstrap": {  

    "mappings": {  

        "dispatcherHandlers": {}  

    },  

    "parentId": null  

}  

}  

}
}
```

For the sake of brevity, this response has been abridged to only show a single request handler. Specifically, it shows that GET requests for /ingredients will be handled by the `allIngredients()` method of `IngredientsController`.

MANAGING LOGGING LEVELS

Logging is an important feature of any application. Logging can provide a means of auditing as well as a crude means of debugging.

Setting logging levels can be quite a balancing act. If you set the logging level to be too verbose, there may be too much noise in the logs, and finding useful information may be difficult. On the other hand, if you set logging levels to be too slack, the logs may not be of much value in understanding what an application is doing.

Logging levels are typically applied on a package-by-package basis. If you're ever wondering what logging levels are set in your running Spring Boot application, you can issue a GET request to the /loggers endpoint. The following JSON shows an excerpt from a response to /loggers:

```
{
  "levels": [ "OFF", "ERROR", "WARN", "INFO", "DEBUG", "TRACE" ],
  "loggers": {
    "ROOT": {
      "name": "ROOT"
    }
  }
}
```

```
        "configuredLevel": "INFO", "effectiveLevel": "INFO"
    },
    ...
    "org.springframework.web": {
        "configuredLevel": null, "effectiveLevel": "INFO"
    },
    ...
    "tacos": {
        "configuredLevel": null, "effectiveLevel": "INFO"
    },
    "tacos.ingredients": {
        "configuredLevel": null, "effectiveLevel": "INFO"
    },
    "tacos.ingredients.IngredientServiceApplication": {
        "configuredLevel": null, "effectiveLevel": "INFO"
    }
}
```

The response starts off with a list of all valid logging levels. After that, the loggers element lists logging-level details for each package in the application. The configuredLevel property shows the logging level that has been explicitly configured (or null if it hasn't been explicitly configured). The effectiveLevel property gives the effective logging level, which may have been inherited from a parent package or from the root logger.

Although this excerpt only shows logging levels for the root logger and four packages, the complete response will include logging-level entries for every single package in the application, including those for libraries that are in use. If you'd rather focus your request on a specific package, you can specify the package name as an extra path component in the request.

For example, if you just want to know what logging levels are set for the `taco-cloud.ingredients` package, you can make a request to `/loggers/tacos.ingredients`:

```
{  
    "configuredLevel": null,  
    "effectiveLevel": "INFO"  
}
```

Aside from returning the logging levels for the application packages, the `/loggers` endpoint also allows you to change the configured logging level by issuing a POST request. For example, suppose you want to set the logging level of the `taco-cloud.ingredients` package to `DEBUG`. The following curl command will achieve that:

```
$ curl localhost:8081/actuator/loggers/tacos/ingredients \
  -d'{"configuredLevel": "DEBUG"}' \
  -H"Content-type: application/json"
```

Now that the logging level has been changed, you can issue a GET request to `/loggers/tacos/ingredients` to see that it has been changed:

```
{  
  "configuredLevel": "DEBUG",  
  "effectiveLevel": "DEBUG"  
}
```

Notice that where the `configuredLevel` was previously null, it's now `DEBUG`. That change carries over to the `effectiveLevel` as well. But what's most important is that if any code in that package logs anything at debug level, the log files will include that debug-level information.

16.2.3 Viewing application activity

It can be useful to keep an eye on activity in a running application, including the kinds of HTTP requests that the application is handling and the activity of all of the threads in the application. For this, Actuator provides the `/httptrace`, `/threaddump`, and `/heapdump` endpoints.

The `/heapdump` endpoint is perhaps the most difficult Actuator endpoint to describe in any detail. Put succinctly, it downloads a gzip-compressed HPROF heap dump file that can be used to track down memory or thread issues. For the sake of space and because use of the heap dump is a rather advanced feature, I'm going to limit coverage of the `/heapdump` endpoint to this paragraph.

TRACING HTTP ACTIVITY

The `/httptrace` endpoint reports details on the most recent 100 requests handled by an application. Details included are the request method and path, a timestamp indicating when the request was handled, headers from both the request and the response, and the time taken handling the request.

The following snippet of JSON shows a single entry from the response of the `/httptrace` endpoint:

```
{  
  "traces": [  
    {  
      "timestamp": "2018-06-03T23:41:24.494Z",  
      "principal": null,  
      "session": null,  
      "request": {  
        "method": "GET",  
        "uri": "http://localhost:8081/ingredients",  
        "headers": {  
          "Host": ["localhost:8081"],  
          "User-Agent": ["curl/7.54.0"],  
          "Accept": ["*/*"]  
        },  
        "remoteAddress": null  
      },  
      "response": {  
        "status": 200,  
        "headers": {  
          "Content-Type": ["application/json; charset=UTF-8"]  
        }  
      }  
    }  
  ]  
}
```

```

        }
    },
    "timeTaken": 4
},
...
]
}

```

Although this information may be useful for debugging purposes, it's even more interesting when the trace data is tracked over time, providing insight into how busy the application was at any given time as well as how many requests were successful compared to how many failed, based on the value of the response status. In chapter 17, you'll see how Spring Boot Admin captures this information into a running graph that visualizes the HTTP trace information over a period of time.

MONITORING THREADS

In addition to HTTP request tracing, thread activity can also be useful in determining what's going on in a running application. The `/threaddump` endpoint produces a snapshot of current thread activity. The following snippet from a `/threaddump` response gives a taste of what this endpoint provides:

```

{
  "threadName": "reactor-http-nio-8",
  "threadId": 338,
  "blockedTime": -1,
  "blockedCount": 0,
  "waitedTime": -1,
  "waitedCount": 0,
  "lockName": null,
  "lockOwnerId": -1,
  "lockOwnerName": null,
  "inNative": true,
  "suspended": false,
  "threadState": "RUNNABLE",
  "stackTrace": [
    {
      "methodName": "kevent0",
      "fileName": "KQueueArrayWrapper.java",
      "lineNumber": -2,
      "className": "sun.nio.ch.KQueueArrayWrapper",
      "nativeMethod": true
    },
    {
      "methodName": "poll",
      "fileName": "KQueueArrayWrapper.java",
      "lineNumber": 198,
      "className": "sun.nio.ch.KQueueArrayWrapper",
      "nativeMethod": false
    },
    ...
  ],
}

```

```

"lockedMonitors": [
  {
    "className": "io.netty.channel.nio.SelectedSelectionKeySet",
    "identityHashCode": 1039768944,
    "lockedStackDepth": 3,
    "lockedStackFrame": {
      "methodName": "lockAndDoSelect",
      "fileName": "SelectorImpl.java",
      "lineNumber": 86,
      "className": "sun.nio.ch.SelectorImpl",
      "nativeMethod": false
    }
  },
  ...
],
"lockedSynchronizers": [],
"lockInfo": null
}

```

The complete thread dump report includes every thread in the running application. To save space, the thread dump here shows an abridged entry for a single thread. As you can see, it includes details regarding the blocking and locking status of the thread, among other thread specifics. There's also a stack trace that gives some insight into which area of the code the thread is spending time on.

Since the /threaddump endpoint only provides a snapshot of thread activity at the time it was requested, it can be difficult to get a full picture of how threads are behaving over time. In chapter 17, you'll see how Spring Boot Admin can monitor the /threaddump endpoint in a live view.

16.2.4 Tapping runtime metrics

The /metrics endpoint is capable of reporting all manner of metrics produced by a running application, including metrics concerning memory, processor, garbage collection, and HTTP requests. There are over two dozen categories of metrics provided out of the box by Actuator, as evidenced by the list of metrics categories returned when issuing a GET request to /metrics:

```
$ curl localhost:8081/actuator/metrics | jq
{
  "names": [
    "jvm.memory.max",
    "process.files.max",
    "jvm.gc.memory.promoted",
    "http.server.requests",
    "system.load.average.1m",
    "jvm.memory.used",
    "jvm.gc.max.data.size",
    "jvm.memory.committed",
    "system.cpu.count",
    "logback.events",
    "jvm.buffer.memory.used",
    "jvm.threads.daemon",
    "jvm.threads.blocked"
  ]
}
```

```

    "system.cpu.usage",
    "jvm.gc.memory.allocated",
    "jvm.threads.live",
    "jvm.threads.peak",
    "process.uptime",
    "process.cpu.usage",
    "jvm.classes.loaded",
    "jvm.gc.pause",
    "jvm.classes.unloaded",
    "jvm.gc.live.data.size",
    "process.files.open",
    "jvm.buffer.count",
    "jvm.buffer.total.capacity",
    "process.start.time"
]
}

```

There are so many metrics covered that it would be impossible to discuss them all in any meaningful way in this chapter. Instead, let's focus on one category of metrics, `http.server.requests`, as an example of how to consume the `/metrics` endpoint.

If instead of simply requesting `/metrics`, you were to issue a GET request for `/metrics/{METRICS CATEGORY}`, you'd receive more detail about the metrics for that category. In the case of `http.server.requests`, a GET request for `/metrics/ http.server.requests` returns data that looks like the following:

```

$ curl localhost:8081/actuator/metrics/http.server.requests
{
  "name": "http.server.requests",
  "measurements": [
    { "statistic": "COUNT", "value": 2103 },
    { "statistic": "TOTAL_TIME", "value": 18.086334315 },
    { "statistic": "MAX", "value": 0.028926313 }
  ],
  "availableTags": [
    { "tag": "exception",
      "values": [ "ResponseStatusException",
                  "IllegalArgumentException", "none" ] },
    { "tag": "method", "values": [ "GET" ] },
    { "tag": "uri",
      "values": [
        "/actuator/metrics/{requiredMetricName}",
        "/actuator/health", "/actuator/info", "/ingredients",
        "/actuator/metrics", "/*" ] },
    { "tag": "status", "values": [ "404", "500", "200" ] }
  ]
}

```

The most significant portion of this response is the `measurements` section, which includes all the metrics for the requested category. In this case, it reports that there have been 2,103 HTTP requests. The total time spent handling those requests is 18.086334315 seconds, and the maximum time spent processing any request is 0.028926313 seconds.

Those generic metrics are interesting, but you can narrow down the results further by using the tags listed under availableTags. For example, you know that there have been 2,103 requests, but what's unknown is how many of them resulted in an HTTP 200 versus an HTTP 404 or HTTP 500 response status. Using the status tag, you can get metrics for all requests resulting in an HTTP 404 status like this:

```
$ curl localhost:8081/actuator/metrics/http.server.requests? \
    tag=status:404
{
  "name": "http.server.requests",
  "measurements": [
    { "statistic": "COUNT", "value": 31 },
    { "statistic": "TOTAL_TIME", "value": 0.522061212 },
    { "statistic": "MAX", "value": 0 }
  ],
  "availableTags": [
    { "tag": "exception",
      "values": [ "ResponseStatusException", "none" ] },
    { "tag": "method", "values": [ "GET" ] },
    { "tag": "uri",
      "values": [
        "/actuator/metrics/{requiredMetricName}", "/**" ] }
  ]
}
```

By specifying the tag name and value with the tag request attribute, you now see metrics specifically for requests that resulted in an HTTP 404 response. This shows that there were 31 requests resulting in a 404, and it took 0.522061212 seconds to serve them all. Moreover, it's clear that some of the failing requests were GET requests for /actuator/metrics/{requiredMetricsName} (although it's unclear what the {requiredMetricsName} path variable resolved to). And some were for some other path, captured by the /** wildcard path.

Hmmm ... What if you want to know how many of those HTTP 404 responses were for the /** path? All you need to do to filter this further is to specify the uri tag in the request, like this:

```
% curl "localhost:8081/actuator/metrics/http.server.requests? \
    tag=status:404&tag=uri:/**"
{
  "name": "http.server.requests",
  "measurements": [
    { "statistic": "COUNT", "value": 30 },
    { "statistic": "TOTAL_TIME", "value": 0.519791548 },
    { "statistic": "MAX", "value": 0 }
  ],
  "availableTags": [
    { "tag": "exception", "values": [ "ResponseStatusException" ] },
    { "tag": "method", "values": [ "GET" ] }
  ]
}
```

Now you can see that there were 30 requests for some path that matched `/**` that resulted in an HTTP 404 response, and it took a total of 0.519791548 seconds to handle those requests.

You'll also notice that as you refine the request, the available tags are more limited. The tags offered are only those that match the requests captured by the displayed metrics. In this case, the `exception` and `method` tags each only have a single value; it's obvious that all 30 of the requests were GET requests that resulted in a 404 because of a `ResponseStatusException`.

Navigating the `/metrics` endpoint can be a tricky business, but with a little practice, it's not impossible to get the data you're looking for. In chapter 17, you'll see how Spring Boot Admin makes consuming data from the `/metrics` endpoint much easier.

Although the information presented by Actuator endpoints offers useful insight into the inner workings of a running Spring Boot application, they're not well suited for human consumption. As REST endpoints, they're intended for consumption by some other application, perhaps a UI. With that in mind, let's see how you can present Actuator information in a user-friendly web application.

16.3 Customizing Actuator

One of the greatest features of Actuator is that it can be customized to meet the specific needs of an application. A few of the endpoints themselves allow for customization. Meanwhile, Actuator itself allows you to create custom endpoints.

Let's look at a few ways that Actuator can be customized, starting with ways to add information to the `/info` endpoint.

16.3.1 Contributing information to the `/info` endpoint

As you saw in section 16.2.1, the `/info` endpoint starts off empty and uninformative. But you can easily add data to it by creating properties that are prefixed with `info..`

While prefixing properties with `info..` is a very easy way to get custom data into the `/info` endpoint, it's not the only way. Spring Boot offers an interface named `InfoContributor` that allows you to programmatically add any information you want to the `/info` endpoint response. Spring Boot even comes ready with a couple of useful implementations of `InfoContributor` that you'll no doubt find useful.

Let's see how you can write your own `InfoContributor` to add some custom info to the `/info` endpoint.

CREATING A CUSTOM INFO CONTRIBUTOR

Suppose you want to add some simple statistics regarding Taco Cloud to the `/info` endpoint. For example, let's say you want to include information about how many tacos have been created. To do that, you can write a class that implements `InfoContributor`, inject it with `TacoRepository`, and then publish whatever count that `TacoRepository` gives you as information to the `/info` endpoint. The next listing shows how you might implement such a contributor.

Listing 16.3 A custom implementation of InfoContributor

```
package tacos.tacos;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;
import java.util.HashMap;
import java.util.Map;
import org.springframework.boot.actuate.info.Info.Builder;

@Component
public class TacoCountInfoContributor implements InfoContributor {
    private TacoRepository tacoRepo;

    public TacoCountInfoContributor(TacoRepository tacoRepo) {
        this.tacoRepo = tacoRepo;
    }

    @Override
    public void contribute(Builder builder) {
        long tacoCount = tacoRepo.count();
        Map<String, Object> tacoMap = new HashMap<String, Object>();
        tacoMap.put("count", tacoCount);
        builder.withDetail("taco-stats", tacoMap);
    }
}
```

By implementing `InfoContributor`, `TacoCountInfoContributor` is required to implement the `contribute()` method. This method is given a `Builder` object on which the `contribute()` method makes a call to `withDetail()` to add info details. In your implementation, you consult `TacoRepository` by calling its `count()` method to find out how many tacos have been created. Then you put that count into a `Map`, which you then give to the builder with the label `taco-stats`. The results of the `/info` endpoint will include that count, as shown here:

```
{
    "taco-stats": {
        "count": 44
    }
}
```

As you can see, an implementation of `InfoContributor` is able to use whatever means necessary to contribute information. This is in contrast to simply prefixing a property with `info.`, which, while simple, is limited to static values.

INJECTING BUILD INFORMATION INTO THE /INFO ENDPOINT

Spring Boot comes with a few built-in implementations of `InfoContributor` that automatically add information to the results of the `/info` endpoint. Among them is `BuildInfoContributor`, which adds information from the project build file into the `/info` endpoint results. This includes basic information such as the project version, the timestamp of the build, and the host and user who performed the build.

To enable build information to be included in the results of the /info endpoint, add the build-info goal to the Spring Boot Maven Plugin executions, as follows:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>build-info</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

If you're using Gradle to build your project, you can simply add the following lines to your build.gradle file:

```
springBoot {
  buildInfo()
}
```

In either event, the build will produce a file named build-info.properties in the distributable JAR or WAR file that BuildInfoContributor will consume and contribute to the /info endpoint. The following snippet from the /info endpoint response shows the build information that's contributed:

```
{
  "build": {
    "version": "0.0.16-SNAPSHOT",
    "artifact": "ingredient-service",
    "name": "ingredient-service",
    "group": "sia5",
    "time": "2018-06-04T00:24:04.373Z"
  }
}
```

This information is useful for understanding exactly which version of an application is running and when it was built. By performing a GET request to the /info endpoint, you'll know whether or not you're running the latest and greatest build of the project.

EXPOSING GIT COMMIT INFORMATION

Assuming that your project is kept in Git for source code control, you may want to include Git commit information in the /info endpoint. To do that, you'll need to add the following plugin in the Maven project pom.xml:

```
<build>
  <plugins>
    ...
      <plugin>
        <groupId>pl.project13.maven</groupId>
        <artifactId>git-commit-id-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
```

If you're a Gradle user, don't worry. There's an equivalent plugin for you to add to your build.gradle file:

```
plugins {
  id "com.gorylenko.gradle-git-properties" version "1.4.17"
}
```

Both of these plugins do essentially the same thing: they generate a build-time artifact named git.properties that contains all of the Git metadata for the project. A special InfoContributor implementation discovers that file at runtime and exposes its contents as part of the /info endpoint.

In its simplest form, the Git information presented in the /info endpoint includes the Git branch, commit hash, and timestamp that the application was built against:

```
{
  "git": {
    "commit": {
      "time": "2018-06-02T18:10:58Z",
      "id": "b5c104d"
    },
    "branch": "master"
  },
  ...
}
```

This information is quite definitive in describing the state of the code when the project was built. But by setting the management.info.git.mode property to full

```
management:
  info:
    git:
      mode: full
```

you can get extremely detailed information about the Git commit that was in play when the project was built. The following listing shows a sample of what the full Git info might look like.

Listing 16.4 Full Git commit info exposed through the /info endpoint

```
{
  "git": {
    "build": {
```

```
"host": "DarkSide.local",
"version": "0.0.16-SNAPSHOT",
"time": "2018-06-02T18:11:23Z",
"user": {
    "name": "Craig Walls",
    "email": "craig@habuma.com"
},
"branch": "master",
"commit": {
    "message": {
        "short": "Add Spring Boot Admin and Actuator",
        "full": "Add Spring Boot Admin and Actuator"
    },
    "id": {
        "describe": "b5c104d-dirty",
        "abbrev": "b5c104d",
        "describe-short": "b5c104d-dirty",
        "full": "b5c104d1fcbe6c2b84965ea08a330595100fd44e"
    },
    "time": "2018-06-02T18:10:58Z",
    "user": {
        "email": "craig@habuma.com",
        "name": "Craig Walls"
    }
},
"closest": {
    "tag": {
        "name": "",
        "commit": {
            "count": ""
        }
    }
},
"dirty": "true",
"remote": {
    "origin": {
        "url": "Unknown"
    }
},
"tags": ""
},
...
}
```

In addition to the timestamp and abbreviated Git commit hash, the full version includes the name and email of the user who committed the code as well as the commit message and other information, allowing you to pinpoint exactly what code was used to build the project. In fact, notice that the `dirty` field in listing 16.4 is `true`, indicating that there were some uncommitted changes in the build directory when the project was built. It doesn't get much more damning than that!

16.3.2 Defining custom health indicators

Spring Boot comes with several out-of-the-box health indicators that provide health information for many common external systems that a Spring application may integrate with. But at some point, you may find that you're interacting with some external system that Spring Boot never anticipated, nor provided a health indicator for.

For instance, your application may integrate with a legacy mainframe application, and the health of your application may be affected by the health of the legacy system. To create a custom health indicator, all you need to do is create a bean that implements the `HealthIndicator` interface.

As it turns out, the Taco Cloud services have no need for a custom health indicator, as the ones provided by Spring Boot are more than sufficient. But to demonstrate how you can develop a custom health indicator, consider the next listing, which shows a simple implementation of `HealthIndicator` in which health is determined somewhat randomly by the time of day.

Listing 16.5 An unusual implementation of `HealthIndicator`

```
package tacos.tacos;
import java.util.Calendar;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class WackoHealthIndicator
    implements HealthIndicator {
    @Override
    public Health health() {
        int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
        if (hour > 12) {
            return Health
                .outOfService()
                .withDetail("reason",
                    "I'm out of service after lunchtime")
                .withDetail("hour", hour)
                .build();
        }

        if (Math.random() < 0.1) {
            return Health
                .down()
                .withDetail("reason", "I break 10% of the time")
                .build();
        }

        return Health
            .up()
            .withDetail("reason", "All is good!")
            .build();
    }
}
```

This crazy health indicator first checks what the current time is, and if it's after noon, returns a health status of OUT_OF_SERVICE, with a few details explaining the reason for that status.

Even if it's before lunch, there's a 10% chance that the health indicator will report a DOWN status, because it uses a random number to decide if it's up or not. If the random number is less than 0.1, the status will be reported as DOWN. Otherwise, the status will be UP.

Obviously, the health indicator in listing 16.5 isn't going to be very useful in any real-world applications. But imagine that instead of consulting the current time or a random number, it were to make a remote call to some external system and determine the status based on the response it receives. In that case, it would be a very useful health indicator.

16.3.3 Registering custom metrics

In section 16.2.4, we looked at how you could navigate the /metrics endpoint to consume various metrics published by Actuator, with a focus on metrics pertaining to HTTP requests. The metrics provided by Actuator are very useful, but the /metrics endpoint isn't limited to only those built-in metrics.

Ultimately, Actuator metrics are implemented by Micrometer (<https://micrometer.io/>), a vendor-neutral metrics facade that makes it possible for applications to publish any metrics they want and to display them in the third-party monitoring system of their choice, including support for Prometheus, Datadog, and New Relic, among others.

The most basic means of publishing metrics with Micrometer is through Micrometer's MeterRegistry. In a Spring Boot application, all you need to do to publish metrics is to inject a MeterRegistry wherever you may need to publish counters, timers, or gauges that capture the metrics for your application.

As an example of publishing custom metrics, suppose you want to keep counters for the numbers of tacos that have been created with different ingredients. That is, you want to track how many tacos have been made with lettuce, or ground beef, or flour tortillas, or any of the available ingredients. The TacoMetrics bean in the next listing shows how you might use MeterRegistry to gather that information.

Listing 16.6 TacoMetrics registers metrics around taco ingredients

```
package tacos.tacos;
import java.util.List;
import
    org.springframework.data.rest.core.event.AbstractRepositoryEventListener
;
import org.springframework.stereotype.Component;
import io.micrometer.core.instrument.MeterRegistry;

@Component
public class TacoMetrics extends AbstractRepositoryEventListener<Taco> {
    private MeterRegistry meterRegistry;
```

```

public TacoMetrics(MeterRegistry meterRegistry) {
    this.meterRegistry = meterRegistry;
}

@Override
protected void onAfterCreate(Taco taco) {
    List<Ingredient> ingredients = taco.getIngredients();
    for (Ingredient ingredient : ingredients) {
        meterRegistry.counter("tacocloud",
            "ingredient", ingredient.getId()).increment();
    }
}
}

```

As you can see, `TacoMetrics` is injected through its constructor with a `MeterRegistry`. It also extends `AbstractRepositoryEventListener`, a Spring Data class that enables the interception of repository events and overrides the `onAfterCreate()` method so that it can be notified any time a new `Taco` object is saved.

Within `onAfterCreate()`, a counter is declared for each ingredient where the tag name is `ingredient` and the tag value is equal to the ingredient ID. If a counter with that tag already exists, it will be reused. The counter is incremented, indicating that another `taco` has been created for the ingredient.

After a few tacos have been created, you can start querying the `/metrics` endpoint for ingredient counts. A GET request to `/metrics/tacocloud` yields some unfiltered metric counts:

```

$ curl localhost:8087/actuator/metrics/tacocloud
{
  "name": "tacocloud",
  "measurements": [ { "statistic": "COUNT", "value": 84 } ],
  "availableTags": [
    {
      "tag": "ingredient",
      "values": [ "FLTO", "CHED", "LETC", "GRBF",
                  "COTO", "JACK", "TMTO", "SLSA" ]
    }
  ]
}

```

The count value under `measurements` doesn't mean much here, as it's a sum of all the counts for all ingredients. But let's suppose you want to know how many tacos have been created with flour tortillas (`FLTO`). All you need to do is specify the `ingredient` tag with a value of `FLTO`:

```

$ curl localhost:8087/actuator/metrics/tacocloud?tag=ingredient:FLTO
{
  "name": "tacocloud",
  "measurements": [

```

```

        { "statistic": "COUNT", "value": 39 }
    ],
    "availableTags": []
}

```

Now it's clear that 39 tacos have had flour tortillas as one of their ingredients.

16.3.4 Creating custom endpoints

At first glance, you might think that Actuator's endpoints are implemented as nothing more than Spring MVC controllers. But as you'll see in chapter 18, the endpoints are also exposed as JMX MBeans as well as through HTTP requests. Therefore, there must be something more to these endpoints than just a controller class.

In fact, Actuator endpoints are defined quite differently from controllers. Instead of a class that's annotated with `@Controller` or `@RestController`, Actuator endpoints are defined with classes that are annotated with `@Endpoint`.

What's more, instead of using HTTP-named annotations such as `@GetMapping`, `@PostMapping`, or `@DeleteMapping`, Actuator endpoint operations are defined by methods annotated with `@ReadOperation`, `@WriteOperation`, and `@DeleteOperation`. These annotations don't imply any specific communication mechanism and, in fact, allow Actuator to communicate by any variety of communication mechanisms, HTTP, and JMX out of the box.

To demonstrate how to write a custom Actuator endpoint, consider `NotesEndpoint`.

Listing 16.7 A custom endpoint for taking notes

```

package tacos.ingredients;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import org.springframework.boot.actuate.endpoint.annotation.DeleteOperation;
import org.springframework.boot.actuate.endpoint.annotation.Endpoint;
import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;
import org.springframework.boot.actuate.endpoint.annotation.WriteOperation;
import org.springframework.stereotype.Component;
import lombok.Getter;
import lombok.RequiredArgsConstructor;

@Component
@Endpoint(id="notes", enableByDefault=true)
public class NotesEndpoint {

    private List<Note> notes = new ArrayList<>();

    @ReadOperation
    public List<Note> notes() {
        return notes;
    }
}

```

```
@WriteOperation
public List<Note> addNote(String text) {
    notes.add(new Note(text));
    return notes;
}

@DeleteOperation
public List<Note> deleteNote(int index) {
    if (index < notes.size()) {
        notes.remove(index);
    }
    return notes;
}

@RequiredArgsConstructor
private class Note {
    @Getter
    private Date time = new Date();

    @Getter
    private final String text;
}
}
```

This endpoint is a simple note-taking endpoint, wherein one can submit a note with a write operation, read the list of notes with a read operation, and remove a note with the delete operation. Admittedly, this endpoint isn't very useful as far as Actuator endpoints go. But when you consider that the out-of-the-box Actuator endpoints cover so much ground, it's difficult to come up with a practical example of a custom Actuator endpoint.

At any rate, the `NotesEndpoint` class is annotated with `@Component` so that it will be picked up by Spring's component scanning and instantiated as a bean in the Spring application context. But more relevant to this discussion, it's also annotated with `@Endpoint`, making it an Actuator endpoint with an ID of `notes`. And it's enabled by default so that you won't need to explicitly enable it by including it in the `management.web.endpoints.web.exposure.include` configuration property.

As you can see, `NotesEndpoint` offers one of each kind of operation:

- The `notes()` method is annotated with `@ReadOperation`. When invoked, it will return a list of available notes. In HTTP terms, this means it will handle an HTTP GET request for `/actuator/notes` and respond with a JSON list of notes.
- The `addNote()` method is annotated with `@WriteOperation`. When invoked, it will create a new note from the given text and add it to the list. In HTTP terms, it handles a POST request where the body of the request is a JSON object with a `text` property. It finishes by responding with the current state of the notes list.
- The `deleteNote()` method is annotated with `@DeleteOperation`. When invoked, it will delete the note at the given index. In HTTP terms, this endpoint handles DELETE requests where the index is given as a request parameter.

To see this in action, you can use curl to poke about with this new endpoint. First, add a couple of notes, using two separate POST requests:

```
$ curl localhost:8080/actuator/notes \
-d'{"text":"Bring home milk"}' \
-H"Content-type: application/json"
[{"time":"2018-06-08T13:50:45.085+0000","text":"Bring home milk"}]

$ curl localhost:8080/actuator/notes \
-d'{"text":"Take dry cleaning"}' \
-H"Content-type: application/json"
[{"time":"2018-06-08T13:50:45.085+0000","text":"Bring home milk"}, \
 {"time":"2018-06-08T13:50:48.021+0000","text":"Take dry cleaning"}]
```

As you can see, each time a new note is posted, the endpoint responds with the newly appended list of notes. But if later you want to view the list of notes, you can do a simple GET request:

```
$ curl localhost:8080/actuator/notes
[{"time":"2018-06-08T13:50:45.085+0000","text":"Bring home milk"}, \
 {"time":"2018-06-08T13:50:48.021+0000","text":"Take dry cleaning"}]
```

If you decide to remove one of the notes, a DELETE request with an index request parameter should do the trick:

```
$ curl localhost:8080/actuator/notes?index=1 -X DELETE
[{"time":"2018-06-08T13:50:45.085+0000","text":"Bring home milk"}]
```

It's important to note that although I've only shown how to interact with the endpoint using HTTP, it will also be exposed as an MBean that can be accessed using whatever JMX client you choose. But if you want to limit it to only exposing an HTTP endpoint, you can annotate the endpoint class with `@WebEndpoint` instead of `@Endpoint`:

```
@Component
@WebEndpoint(id="notes", enableByDefault=true)
public class NotesEndpoint {
    ...
}
```

Likewise, if you prefer an MBean-only endpoint, annotate the class with `@JmxEndpoint`.

16.4 Securing Actuator

The information presented by Actuator is probably not something that you would want prying eyes to see. Moreover, because Actuator provides a few operations that let you change environment properties and logging levels, it's probably a good idea to secure Actuator so that only clients with proper access will be allowed to consume its endpoints.

Even though it's important to secure Actuator, security is outside of Actuator's responsibilities. Instead, you'll need to use Spring Security to secure Actuator. And because Actuator endpoints are just paths in the application like any other path in the application, there's nothing unique about securing Actuator versus any other application path. Everything we discussed in chapter 4 applies when securing Actuator endpoints.

Because all Actuator endpoints are gathered under a common base path of /actuator (or possibly some other base path if the `management.endpoints.web.base-path` property is set), it's easy to apply authorization rules to all Actuator endpoints across the board. For example, to require that a user have `ROLE_ADMIN` authority to invoke Actuator endpoints, you might override the `configure()` method of `WebSecurityConfigurerAdapter` like this:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
        .antMatchers("/actuator/**").hasRole("ADMIN")  
  
        .and()  
  
        .httpBasic();  
}
```

This requires that all requests be from an authenticated user with `ROLE_ADMIN` authority. It also configures HTTP basic authentication so that client applications can submit encoded authentication information in their request `Authorization` headers.

The only real problem with securing Actuator this way is that the path to the endpoints is hardcoded as `/actuator/**`. If this were to change because of a change to the `management.endpoints.web.base-path` property, it would no longer work. To help with this, Spring Boot also provides `EndpointRequest`—a request matcher class that makes this even easier and less dependent on a given `String` path. Using `EndpointRequest`, you can apply the same security requirements for Actuator endpoints without hard-coding the `/actuator/**` path:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .requestMatcher(EndpointRequest.toAnyEndpoint())  
        .authorizeRequests()  
        .anyRequest().hasRole("ADMIN")  
        .and()  
        .httpBasic();  
}
```

The `EndpointRequest.toAnyEndpoint()` method returns a request matcher that matches any Actuator endpoint. If you'd like to exclude some of the endpoints from the request matcher, you can call `excluding()`, specifying them by name:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .requestMatcher(
            EndpointRequest.toAnyEndpoint()
                .excluding("health", "info"))
        .authorizeRequests()
            .anyRequest().hasRole("ADMIN")
        .and()
            .httpBasic();
}

```

On the other hand, should you wish to apply security to only a handful of Actuator endpoints, you can specify those endpoints by name by calling `to()` instead of `toAnyEndpoint()`:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .requestMatcher(EndpointRequest.to(
            "beans", "threaddump", "loggers"))
        .authorizeRequests()
            .anyRequest().hasRole("ADMIN")
        .and()
            .httpBasic();
}

```

This limits Actuator security to only the `/beans`, `/threaddump`, and `/loggers` endpoints. All other Actuator endpoints are left wide open.

Summary

- Spring Boot Actuator provides several endpoints, both as HTTP and as JMX MBeans, that let you peek into the inner workings of a Spring Boot application.
- Most Actuator endpoints are disabled by default, but can be selectively exposed by setting `management.endpoints.web.exposure.include` and `management.endpoints.web.exposure.exclude`.
- Some endpoints, such as the `/loggers` and `/env` endpoints, allow for write operations to change a running application's configuration on the fly.
- Details regarding an application's build and Git commit can be exposed in the `/info` endpoint.
- An application's health can be influenced by a custom health indicator, tracking the health of an externally integrated application.
- Custom application metrics can be registered through Micrometer, which affords Spring Boot applications instant integration with several popular metrics engines such as Datadog, New Relic, and Prometheus.
- Actuator web endpoints can be secured using Spring Security, much like any other endpoint in a Spring web application.



Administering Spring

This chapter covers

- Setting up the Spring Boot Admin
- Registering client applications
- Working with Actuator endpoints
- Securing the Admin server

A picture is worth a thousand words (or so they say), and for many application users, a user-friendly web application is worth a thousand API calls. Don't get me wrong, I'm a command-line junkie and a big fan of using curl and HTTPie to consume REST APIs. But, sometimes, manually typing the command line to invoke a REST endpoint and then visually inspecting the results can be less efficient than simply clicking a link and reading the results in a web browser.

In the previous chapter, we explored all of the HTTP endpoints exposed by the Spring Boot Actuator. As HTTP endpoints that return JSON responses, there's no limit to how those can be used. In this chapter, we'll see how to put a front-end user interface (UI) on top of the Actuator to make it easier to use, as well as capture live data that would be difficult to consume from Actuator directly.

17.1 Using the Spring Boot Admin

I've been asked several times if it'd make sense and, if so, how hard it'd be to develop a web application that consumes Actuator endpoints and serves them up in an easy-to-view UI. I respond that it's just a REST API and, therefore, anything is possible. But why bother creating your own UI for the Actuator when the good folks at codecentric AG (<https://www.codecentric.de/>), a software and consulting company based in Germany, have already done the work for you?

The Spring Boot Admin is an administrative frontend web application that makes Actuator endpoints more consumable by humans. It's split into two primary components: the Spring Boot Admin server and its clients. The Admin server collects and displays Actuator data that's fed to it from one or more Spring Boot applications, which are identified as Spring Boot Admin clients, as illustrated in figure 17.1.

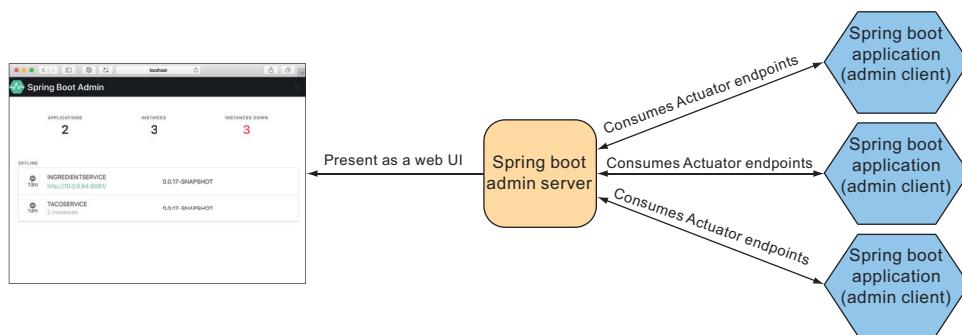


Figure 17.1 The Spring Boot Admin server consumes Actuator endpoints from one or more Spring Boot applications and presents the data in a web-based UI.

You'll need to register each of the applications (the microservices) that make up Taco Cloud as Spring Boot Admin clients. But first, you'll set up the Spring Boot Admin server to receive each client's Actuator information.

17.1.1 Creating an Admin server

To enable the Admin server, you'll first need to create a new Spring Boot application and add the Admin server dependency to the project's build. The Admin server is generally used as a standalone application, separate from any other application. Therefore, the easiest way to get started is to use the Spring Boot Initializr to create a new Spring Boot project and select the checkbox labeled Spring Boot Admin (Server). This results in the following dependency being included in the `<dependencies>` block:

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

Next, you'll need to enable the Admin server by annotating the main configuration class with `@EnableAdminServer` as shown here:

```
package tacos.bootadmin;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import de.codecentric.boot.admin.server.config.EnableAdminServer;

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

Finally, because the Admin server won't be the only application running locally as it's developed, you should set it to listen in on a unique port, but one you can easily access (not port 0, for example). Here, I've chosen port 9090 as the port for the Spring Boot Admin server:

```
server:
  port: 9090
```

NOTE: As with any other service application in a microservice-architected Spring Boot application, the `server.port` property can be set differently in a production profile, where the port may be determined by the underlying platform.

Now your Admin server is ready. If you were to fire it up at this point and navigate to <http://localhost:9090> in your web browser, you'd see something a little like what's shown in figure 17.2.

As you can see, the Spring Boot Admin shows that zero instances of zero applications are all up. But that's meaningless information when you consider the message below those counts that states No Applications Registered. For the Admin server to be useful, you'll need to register some applications with it.

17.1.2 Registering Admin clients

Because the Admin server is an application separate from other Spring Boot application(s) for which it presents Actuator data, you must somehow make the Admin server aware of the applications it should display. Two ways to register Spring Boot Admin clients with the Admin server follow:

- Each application explicitly registers itself with the Admin server.
- The Admin server discovers services through the Eureka service registry.

Let's look at each option, starting with how to configure individual Boot applications as Spring Boot Admin clients so that they can register themselves with the Admin server.

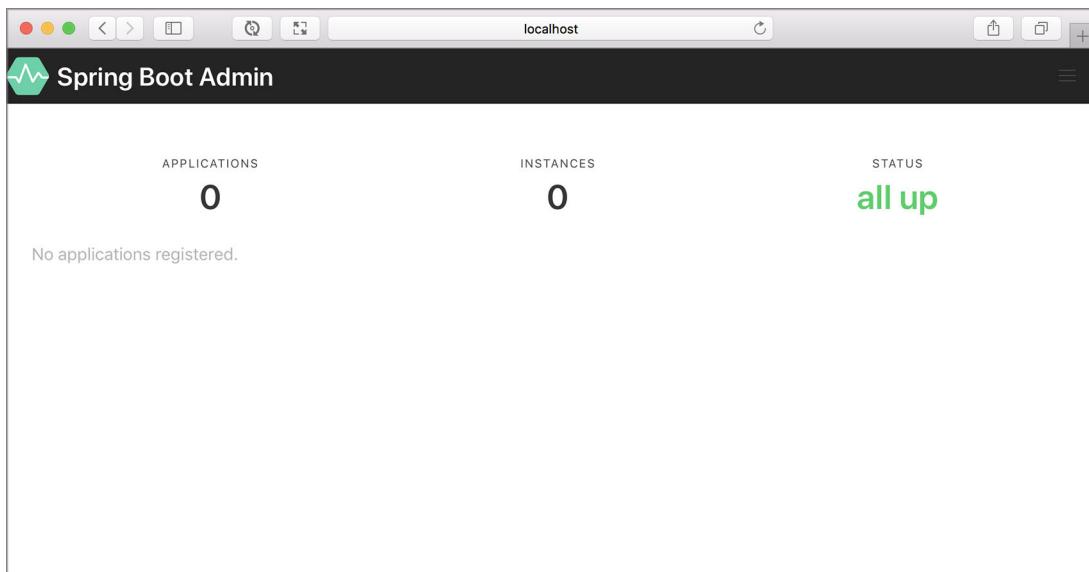


Figure 17.2 A newly created server displayed in the Spring Boot Admin UI. No applications are registered yet.

EXPLICITLY CONFIGURING ADMIN CLIENT APPLICATIONS

In order for a Spring Boot application to register itself as a client of the Admin server, you must include the Spring Boot Admin client starter in its build. You can easily add this dependency to your build by selecting the checkbox labeled Spring Boot Admin (Client) in the Initializr, or you can set the following `<dependency>` for a Maven-built Spring Boot application:

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

With the client-side library in place, you'll also need to configure the location of the Admin server so that the client can register itself. To do that, you'll set the `spring.boot.admin.client.url` property to the root URL of the Admin server:

```
spring:
  application:
    name: ingredient-service
  boot:
    admin:
      client:
        url: http://localhost:9090
```

Notice that the `spring.application.name` property is also set (in this case, for the ingredient service). You've already used this property to identify microservices with

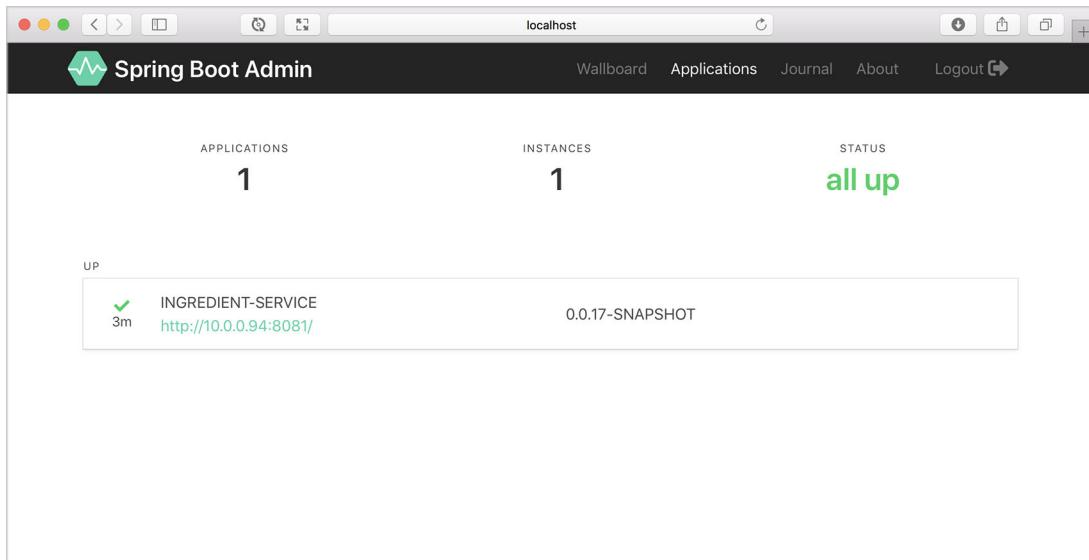


Figure 17.3 The Spring Boot Admin UI displays a single registered application.

the Spring Cloud Config Server and with Eureka. Here it serves a similar purpose: to identify the application to the Admin server. Once you restart the application, you should see it appear in the Admin server as shown in figure 17.3.

Although there isn't much information about the ingredient service shown in figure 17.3, it does show the application's uptime and if the Spring Boot Maven plugin has the build-info goal configured (as we discussed in section 16.3.1), the build version. Rest assured that there are plenty of other runtime details to see after you click the application in the Admin server. We'll look deeper at what the Admin server has to offer in section 17.2.

The same configuration used to register the ingredient service with the Admin server will need to be replicated across all applications. You may find it easier to only configure the `spring.application.name` property and let the Spring Cloud Config Server serve the `spring.boot.admin.client.url` to all of its clients. Or, better yet, if you're already using Eureka as a service registry, then let the Admin server discover services on its own. Let's see how to configure the Admin server as a Eureka client.

DISCOVERING ADMIN CLIENTS

The only thing you must do to enable the Admin server for discovery of services is to add the Spring Cloud Netflix Eureka Client starter to the Admin server's build. Here's the Maven `<dependency>` you'll need:

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

NOTE: You can also get this dependency by checking the Eureka Discovery checkbox in the Spring Initializr.

If the Admin server is enabled as a Eureka client, then nothing else is required. You can skip all of the client configurations described in the previous section because the Admin server automatically discovers all applications registered in Eureka and makes their Actuator data available for display. For example, if there are several of the Taco Cloud services registered in Eureka, then they'll also be shown in the Admin server (figure 17.4).

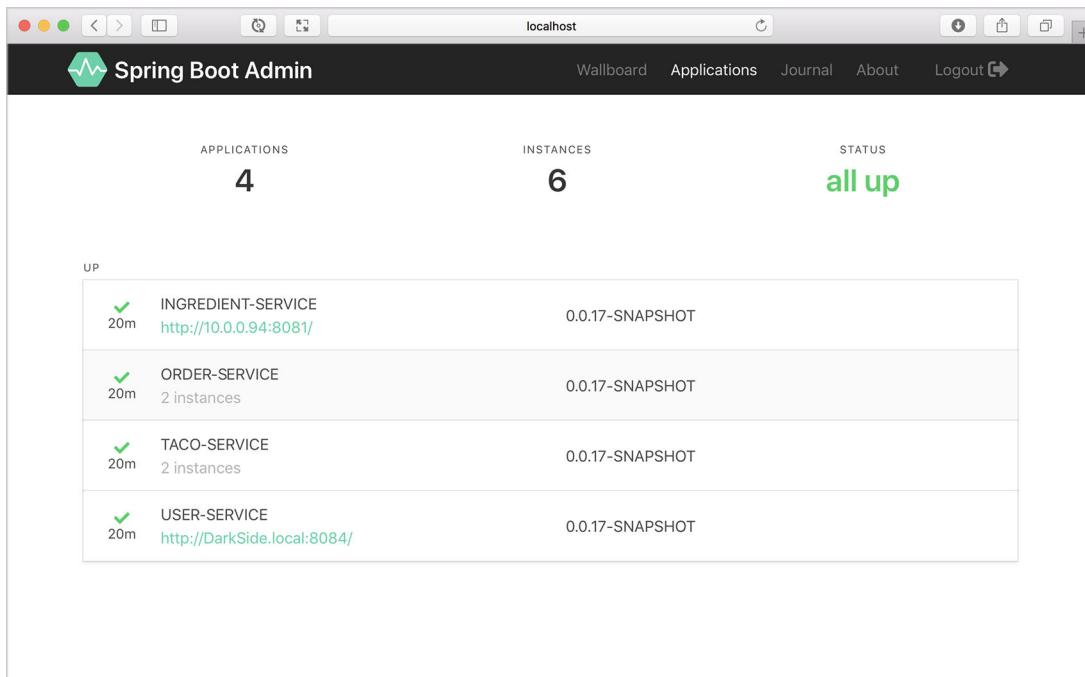


Figure 17.4 The Spring Boot Admin UI can show all the services it discovers in Eureka.

As you can see in figure 17.4, there are four distinct applications listed, as well as a total of six services: two instances of the order service, two instances of the taco service, and one instance of each of the other applications. All applications shown are in the Up status. But if any service goes offline (the user service, for instance), then it may appear separately in the Admin server (figure 17.5).

As a Eureka client, the Admin server also registers itself as a service with Eureka. To keep this from happening, you can set the `eureka.client.register-with-eureka` property to `false`:

```
eureka:  
  client:  
    register-with-eureka: false
```

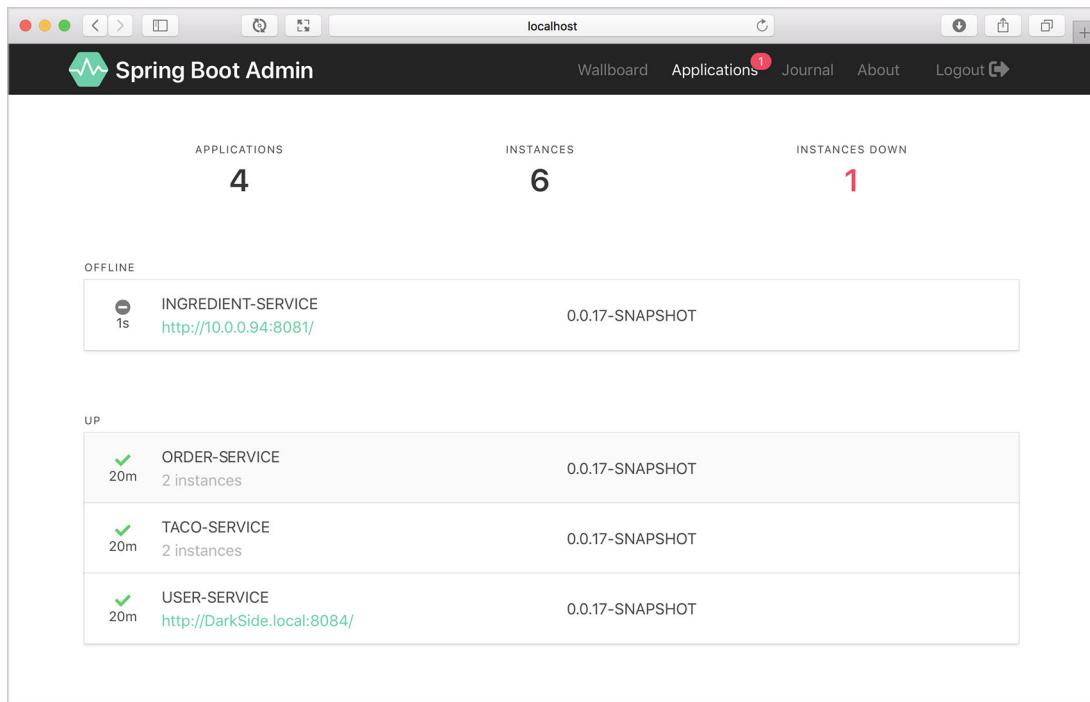


Figure 17.5 The Spring Boot Admin UI shows offline services separately from those that are online.

Like other Eureka clients, you can also configure the location of the Eureka server if it's not listening at the default host and port. The following YAML configures the Eureka location with a host at eureka1.tacocloud.com:

```
eureka:  
  client:  
    service-url:  
      defaultZone: http://eureka1.tacocloud.com:8761/eureka/
```

Now that you have a handful of the Taco Cloud services registered with the Admin server, let's see what the Admin server has to offer.

17.2 Exploring the Admin server

Once you've registered all of the Spring Boot applications as Admin server clients, there's a wealth of information that the Admin server makes available for seeing what's going on inside each application. This includes

- General health and information
- Any metrics published through Micrometer and the /metrics endpoint
- Environment properties

- Logging levels for packages and classes
- Thread tracing details
- HTTP traces for requests
- Audit logs

In fact, almost anything that the Actuator exposes can be viewed in the Admin server, albeit in a much more human-friendly format. This includes graphs and filters to help distill the information. The amount of information presented in the Admin server is far richer than the space we'll have in this chapter to cover it in detail. But let me use the rest of this section to share a few of the highlights of the Admin server.

17.2.1 Viewing general application health and information

As discussed in section 16.2.1, some of the most basic information provided by the Actuator is health and general application information via the /health and /info endpoints. The Admin server displays that information under the Details tab as shown in figure 17.6.

The screenshot shows the Spring Boot Admin UI running on a Mac OS X system. The browser window title is "localhost". The main header bar includes the Spring Boot Admin logo, the URL "localhost", and navigation links for "Wallboard", "Applications", "Journal", "About", and "Logout".

The main content area has a green header bar with the text "INGREDIENT-SERVICE" and "Instance 3be305bc734f (of 1)". To the right of this header are three URLs: "http://DarkSide.local:8081/", "http://DarkSide.local:8081/actuator", and "http://DarkSide.local:8081/actuator/health".

The content is organized into two main sections: "Info" and "Health".

Info Section:

git	commit: time: '2018-06-02T18:10:58Z' id: b5c104d branch: master
build	version: 0.0.17-SNAPSHOT artifact: ingredient-service name: ingredient-service group: tacocloud time: '2018-06-02T18:11:23.515Z'

Metadata Section:

jmx.port	'58589'
management.port	'8081'

Health Section:

Instance	UP
mongo	UP
version	3.2.2

hystrix	UP
diskSpace	UP
total	500 GB
free	174 GB
threshold	10.5 MB

configServer	UP
propertySources	["configClient", "http://github.com/habuma/myapp- config/application.yml (document #0)"]

Figure 17.6 The Details tab of the Spring Boot Admin UI displays general health and information about an application.

If you scroll past the Health and Info sections in the Details tab, you'll find useful statistics from the application's JVM, including graphs displaying memory, thread, and processor usage (figure 17.7).

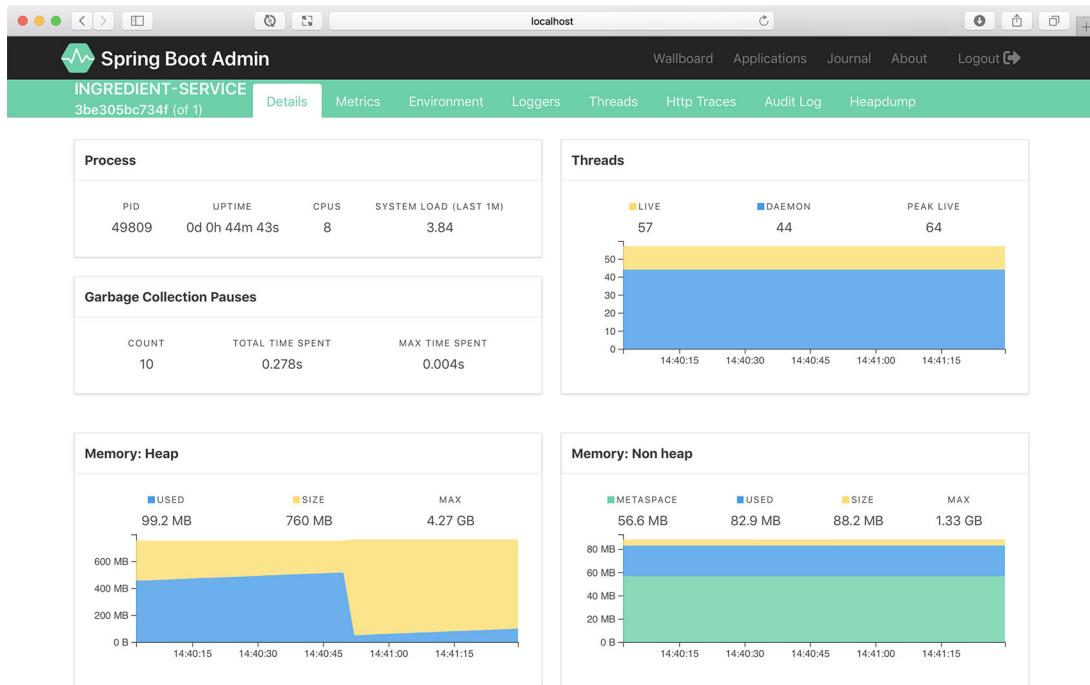


Figure 17.7 As you scroll down on the Details tab, you can view additional JVM internal information, including processor, thread, and memory statistics.

The information displayed in the graphs, as well as the metrics under Processes and Garbage Collection Pauses, can provide useful insights into how your application utilizes JVM resources.

17.2.2 Watching key metrics

The information presented by the /metrics endpoint is perhaps the least human-readable of all of the Actuator's endpoints. But the Admin server makes it easy for us mere mortals to consume the metrics produced in an application with its UI under the Metrics tab.

Initially, the Metrics tab doesn't display any metrics whatsoever. But the form at the top lets you set up one or more watches on any metrics you want to keep an eye on.

In figure 17.8, I've set up two watches on metrics under the http.server.requests category. The first reports metrics anytime an HTTP GET request is received for the /ingredients endpoint; the return status is 200 (OK). The second reports metrics for any request that results in an HTTP 400 (NOT FOUND) response.

The screenshot shows the Spring Boot Admin interface running on localhost. The top navigation bar includes links for Wallboard, Applications, Journal, About, and Logout. Below the header, the title 'INGREDIENT-SERVICE' is displayed, along with the instance identifier 'Instance 3be305bc734f (of 1)'. To the right, three URLs are listed: http://DarkSide.local:8081/, http://DarkSide.local:8081/actuator, and http://DarkSide.local:8081/actuator/health.

The main content area features a tab navigation bar with 'Metrics' selected. Other tabs include Details, Environment, Loggers, Threads, Http Traces, Audit Log, and Heapdump. Below the tabs is a search input field containing 'http.server.requests'. Underneath the search field are four dropdown filters: exception (set to '-'), method (set to '-'), uri (set to '-'), and status (set to '404'). A green 'Add Metric' button is located to the right of these filters.

Below the filters is a table showing metrics data. The columns are labeled 'http.server.requests', 'COUNT', 'TOTAL_TIME', 'MAX', and an empty column. The table contains two rows of data:

http.server.requests	COUNT	TOTAL_TIME	MAX	
exception:none method:GET uri:/ingredients status:200	3428	9.034839572	0.004334177	Delete
status:404	39	0.357279409	0.009219171	Delete

Figure 17.8 On the Metrics tab, you can set up watches on any metrics published through the application's /metrics endpoint.

What's nice about these metrics (and, in fact, almost anything displayed in the Admin server) is that they show live data—they'll automatically update without the need to refresh the page.

17.2.3 Examining environment properties

The Actuator's /env endpoint returns all environment properties available to a Spring Boot application from all of its property sources. And although the JSON response from the endpoint isn't all that difficult to read, the Admin server presents it in a much more aesthetically pleasing form under the Environment tab (figure 17.9).

Because there can be hundreds of properties, you can filter the list of available properties by either property name or value. Figure 17.9 shows properties filtered by those whose name and/or values contain the text "spring.". The Admin server also allows you to set or override environment properties using the form under the Environment Manager header.

The screenshot shows the Spring Boot Admin interface on a Mac OS X system. The title bar says "localhost". The top navigation bar includes "Wallboard", "Applications", "Journal", "About", and "Logout". Below the title bar, the page header is "INGREDIENT-SERVICE" with "Instance 3be305bc734f (of 1)". To the right, there are three URLs: "http://DarkSide.local:8081/", "http://DarkSide.local:8081/actuator", and "http://DarkSide.local:8081/actuator/health". A navigation bar at the top of the main content area includes "Details", "Metrics", "Environment" (which is selected), "Loggers", "Threads", "Http Traces", "Audit Log", and "Heapdump". The main content area is titled "Environment Manager" and contains a search bar for "Property name" and "Value", with "Refresh Context", "Reset", and "Update" buttons. A search input field has "spring.|" typed into it. The "systemProperties" section lists three properties: "spring.beaninfo.ignore" with value "true", "spring.liveBeansView.mbeanDomain", and "spring.application.admin.enabled" with value "true". The "applicationConfig: [classpath:/application.yml]" section shows "spring.application.name" set to "ingredientservice". The "springCloudClientHostInfo" section shows "spring.cloud.client.hostname" as "DarkSide.local" and "spring.cloud.client.ip-address" as "127.0.0.1".

Figure 17.9 The Environment tab displays environment properties and includes options to override and filter those values.

17.2.4 Viewing and setting logging levels

The Actuator's /loggers endpoint is helpful in understanding and overriding logging levels in a running application. The Admin server's Loggers tab adds an easy-to-use UI on top of the /loggers endpoint to make simple work of managing logging in an application. Figure 17.10 shows the list of loggers filtered by the name org.springframework.boot.

By default, the Admin server displays logging levels for all packages and classes. Those can be filtered by name (for classes only) and/or logging levels that are explicitly configured versus inherited from the root logger.

The screenshot shows the Spring Boot Admin UI for the INGREDIENT-SERVICE application. The top navigation bar includes links for Wallboard, Applications, Journal, About, and Logout. Below the header, the application name 'INGREDIENT-SERVICE' and instance ID '3be305bc734f (of 1)' are displayed. A list of loggers is shown, each with a checkbox for 'class only' and 'configured' status, and a dropdown menu for setting logging levels: OFF, ERROR, WARN, INFO (selected), DEBUG, and TRACE. A 'Reset' button is also present. The bottom right corner of the screenshot shows a progress bar indicating '152/699'.

Figure 17.10 The Loggers tab displays logging levels for packages and classes in the application and lets you override those levels.

17.2.5 Monitoring threads

Many threads can run concurrently in any application. Although the /threaddump endpoint (described in section 16.2.3) provides a snapshot of the status of an application's running threads, the Threads tab in the Spring Boot Admin UI keeps a live watch on all of the threads in an application (figure 17.11).

Unlike the /threaddump endpoint that captures a snapshot in time, the bar graphs shown in the Threads tab are constantly updated, showing each thread's status: green if the thread is runnable, yellow if it's waiting, or red if the thread is blocked.

To see more details about an individual thread, click the thread's row in the list. It expands to display historical data about the thread, along with the thread's current stack trace.

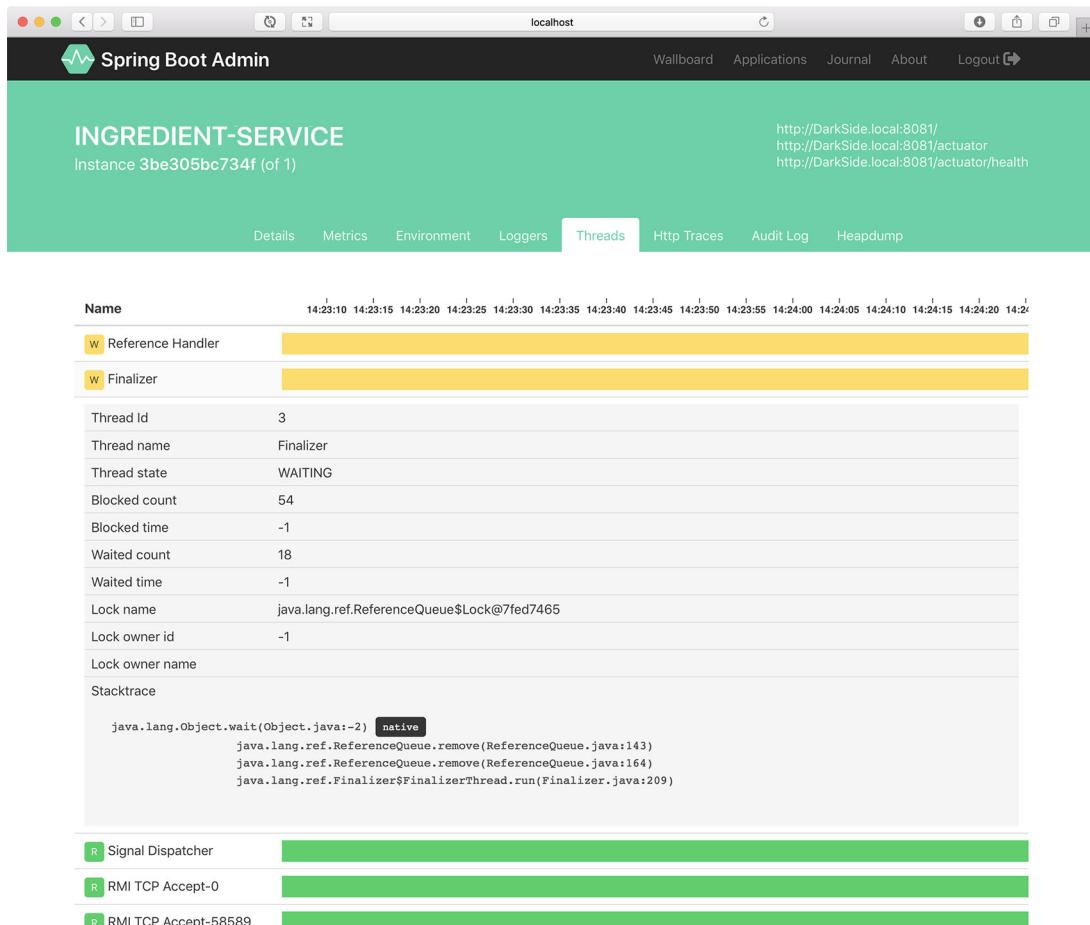


Figure 17.11 You can use the Threads tab in the Admin UI to keep a live watch on an application’s threads.

17.2.6 Tracing HTTP requests

The Spring Boot Admin UI’s HTTP Trace tab (figure 17.12) presents data from the Actuator’s /`httptrace` endpoint. But unlike the /`httptrace` endpoint that returns the 100 most recent HTTP traces at the time of the request, the HTTP Trace tab lists a complete history of HTTP requests. And it’s updated as long as you remain on the tab. If you leave the tab and come back, then it initially only shows the 100 most recent requests, but continues tracing from that point on.

As you can see, the HTTP Trace tab includes a stacked graph at the top that tracks HTTP traffic over time. The graph uses color to indicate successful versus unsuccessful requests. Green indicates success, yellow indicates client errors (for example, 400-level HTTP responses), and red indicates server errors (for example, 500-level HTTP

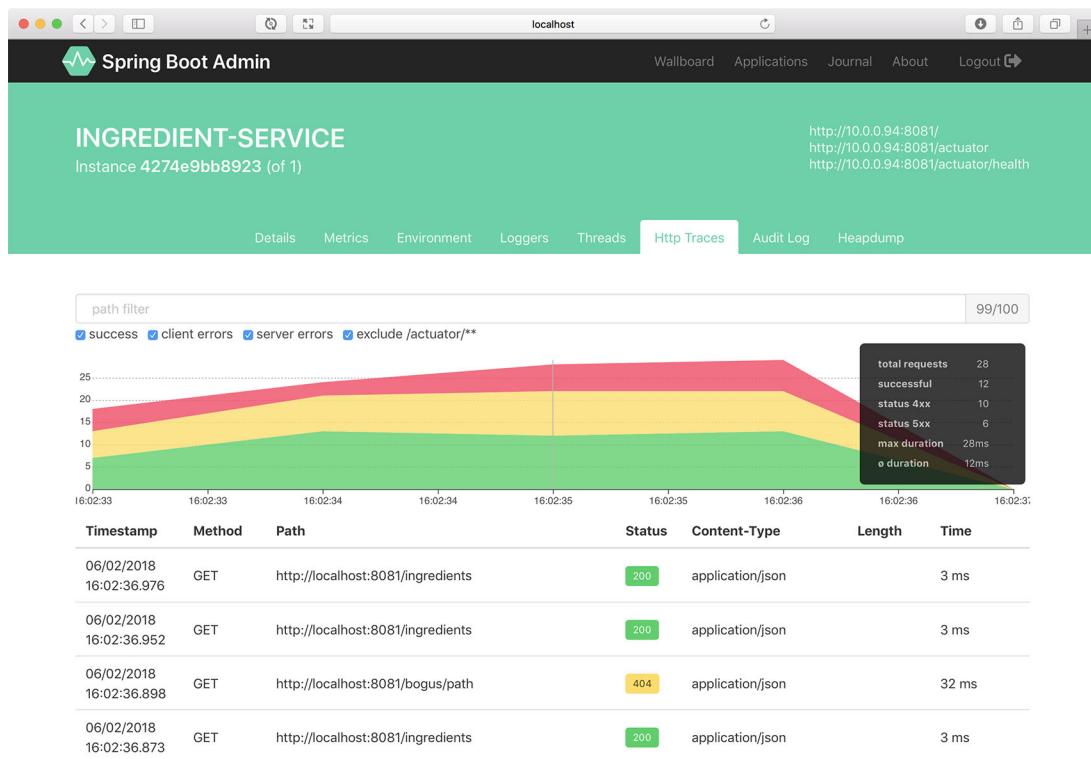


Figure 17.12 The HTTP Trace tab tracks recent HTTP traffic to an application, including information regarding requests that resulted in errors.

responses). If you mouse over the graph, a hover-over box (like the one on the far right of figure 17.12) breaks down the request counts for a given point in time.

Below the graph is the trace history, including a row for each request received by the application. If you click any row, the row expands to show additional information about the request, including any request and response headers (figure 17.13).

17.3 Securing the Admin server

As we discussed in the previous chapter, the information exposed by the Actuator's endpoints isn't intended for general consumption. They contain information that exposes details about an application that only an application administrator should see. Moreover, some of the endpoints allow changes that certainly shouldn't be exposed to just anyone.

Just as security is important to the Actuator, it's also important to the Admin server. What's more, if the Actuator endpoints require authentication, then the Admin server needs to know the credentials to be able to access those endpoints. Let's see how to add a little security to the Admin server. We'll start by requiring authentication.

The screenshot shows a browser window with three tabs: 'multithreading - Is there a goo' (active), 'Spring Boot Admin Reference', and 'Spring Boot Admin'. The URL is 'localhost:9090/#/instances/4274e9bb8923/httptrace'. The main content area is titled 'Spring Boot Admin' and shows the 'INGREDIENT-SERVICE' instance. The 'Http Traces' tab is selected. A single trace entry is listed:

Timestamp	Method	URI	Status	Content-Type	Time Taken
06/02/2018 16:07:57.479	GET	http://localhost:8081/ingredients	200	application/json	2 ms

Clicking on the trace entry reveals detailed request and response logs:

```
{
  "principal": null,
  "session": null,
  "request": {
    "method": "GET",
    "uri": "http://localhost:8081/ingredients",
    "headers": {
      "Host": [
        "localhost:8081"
      ],
      "User-Agent": [
        "curl/7.54.0"
      ],
      "Accept": [
        "*/*"
      ]
    },
    "remoteAddress": null
  },
  "response": {
    "status": 200,
    "headers": {
      "Content-Type": [
        "application/json;charset=UTF-8"
      ]
    }
  },
  "timeTaken": 2,
  "timestamp": "2018-06-02T22:07:57.479Z"
}
```

Figure 17.13 Clicking a request entry on the HTTP Trace tab displays additional details about the request.

17.3.1 Enabling login in the Admin server

It's probably a good idea to add security to the Admin server as it's not secured by default. Because the Admin server is a Spring Boot application, you can secure it using Spring Security just like you would any other Spring Boot application. And just as you would with any application secured by Spring Security, you're free to decide which security scheme fits your needs best.

At a minimum, you can add the Spring Boot security starter to the Admin server's build by checking the Security checkbox in the Initializr or by adding the following <dependency> to the project's pom.xml file:

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

Then, so that you don't have to keep looking at the Admin server's logs for the randomly generated password, you can configure a simple administrative username and password in application.yml:

```
spring:
  security:
    user:
      name: admin
      password: 53cr3t
```

Now when the Admin server is loaded in the browser, you'll be prompted for a username and password with Spring Security's default login form. As in the code snippet, entering admin and 53cr3t will get you in. Of course, this is an extremely basic security configuration. I recommend that you consult chapter 4 for ways of configuring Spring Security for a richer security scheme around the Admin server.

17.3.2 Authenticating with the Actuator

In section 16.4, we discussed how to secure Actuator endpoints with HTTP Basic authentication. By doing so, you'll be able to keep out everyone who doesn't know the username and password you assigned to the Actuator endpoints. Unfortunately, that also means that the Admin server won't be able to consume Actuator endpoints unless it provides the username and password. But how will the Admin server get those credentials?

An Admin server client application can provide its credentials to the Admin server by either registering itself directly with the Admin server or by being discovered through Eureka. If the application registers directly with the Admin server, then it can send its credentials to the server at registration time. You'll need to configure a few properties to enable that.

The `spring.boot.admin.client.instance.metadata.user.name` and `spring.boot.admin.client.instance.metadata.user.password` properties specify the credentials that the Admin server can use to access an application's Actuator endpoints. The following snippet from application.yml shows how you might set those properties:

```
spring:
  boot:
    admin:
      client:
        url: http://localhost:9090
      instance:
        metadata:
          user.name: ${spring.security.user.name}
          user.password: ${spring.security.user.password}
```

The username and password properties must be set in each application that registers itself with the Admin server. The values given must match the username and password that's required in an HTTP Basic authentication header to the Actuator

endpoints. In this example, they're set to `admin` and `password`, which are the credentials configured to access the Actuator endpoints.

On the other hand, if your application is discovered by the Admin server via Eureka, then you'll need to set `eureka.instance.metadata-map.user.name` and `eureka.instance.metadata-map.user.password` instead:

```
eureka:  
  instance:  
    metadata-map:  
      user.name: admin  
      user.password: password
```

When the application registers with Eureka, the credentials will be included in the Eureka registration record's metadata. When the Admin server discovers the application, it fetches the credentials from Eureka, along with other details about the application.

Summary

- The Spring Boot Admin server consumes the Actuator endpoints from one or more Spring Boot applications and presents the data in a user-friendly web application.
- Spring Boot applications can either register themselves as clients to the Admin server or the Admin server can discover them through Eureka.
- Unlike the Actuator endpoints that capture a snapshot of an application's state, the Admin server is able to display a live view into the inner workings of an application.
- The Admin server makes it easy to filter Actuator results and, in some cases, display data visually in a graph.
- Because it's a Spring Boot application, the Admin server can be secured by any means available through Spring Security.

18

Monitoring Spring with JMX

This chapter covers

- Working with Actuator endpoint MBeans
- Exposing Spring beans as MBeans
- Publishing notifications

For over a decade and a half, Java Management Extensions (JMX) has been the standard means of monitoring and managing Java applications. By exposing managed components known as MBeans (managed beans), an external JMX client can manage an application by invoking operations, inspecting properties, and monitoring events from MBeans.

JMX is automatically enabled by default in a Spring Boot application. As a result, all of the Actuator endpoints are exposed as MBeans. And it sets us up nicely to expose any other bean in the Spring application context as an MBean. We'll start exploring Spring and JMX by looking at how Actuator endpoints are exposed as MBeans.

18.1 Working with Actuator MBeans

Take a look back at table 16.1. All of the Actuator endpoints listed there, except for /heapdump, are exposed as MBeans. You can use any JMX client you wish to connect

with Actuator endpoint MBeans. Using JConsole, which comes with the Java Development Kit, you'll find Actuator MBeans listed under the `org.springframework.boot` domain, as shown in figure 18.1.

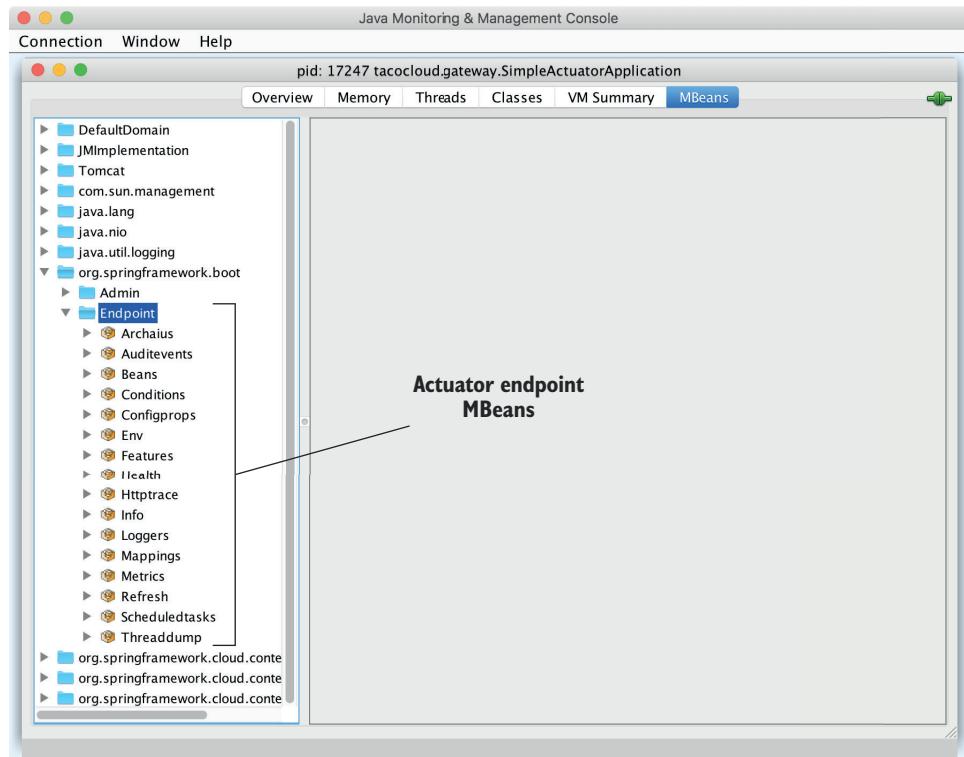


Figure 18.1 Actuator endpoints are automatically exposed as JMX MBeans.

One thing that's nice about Actuator MBean endpoints is that they're all exposed by default. There's no need to explicitly include any of them, as you had to do with HTTP. You can, however, choose to narrow down the choices by setting `management.endpoints.jmx.exposure.include` and `management.endpoints.jmx.exposure.exclude`. For example, to limit Actuator endpoint MBeans to only the `/health`, `/info`, `/bean`, and `/conditions` endpoints, set `management.endpoints.jmx.exposure.include` like this:

```
management:
  endpoints:
    jmx:
      exposure:
        include: health,info,bean,conditions
```

Or, if there are only a few you want to exclude, you can set `management.endpoints.jmx.exposure.exclude` like this:

```
management:
  endpoints:
    jmx:
      exposure:
        exclude: env,metrics
```

Here, you use `management.endpoints.jmx.exposure.exclude` to exclude the `/env` and `/metrics` endpoints. All other Actuator endpoints will still be exposed as MBeans.

To invoke the managed operations on one of the Actuator MBeans in JConsole, expand the endpoint MBean in the left-hand tree, and then select the desired operation under Operations.

For example, if you'd like to inspect the logging levels for the `tacos.ingredients` package, expand the Loggers MBean and click on the operation named `loggerLevels`, as shown in figure 18.2. In the form at the top right, fill in the Name field with the package name (`tacos.ingredients`), and then click the `loggerLevels` button.

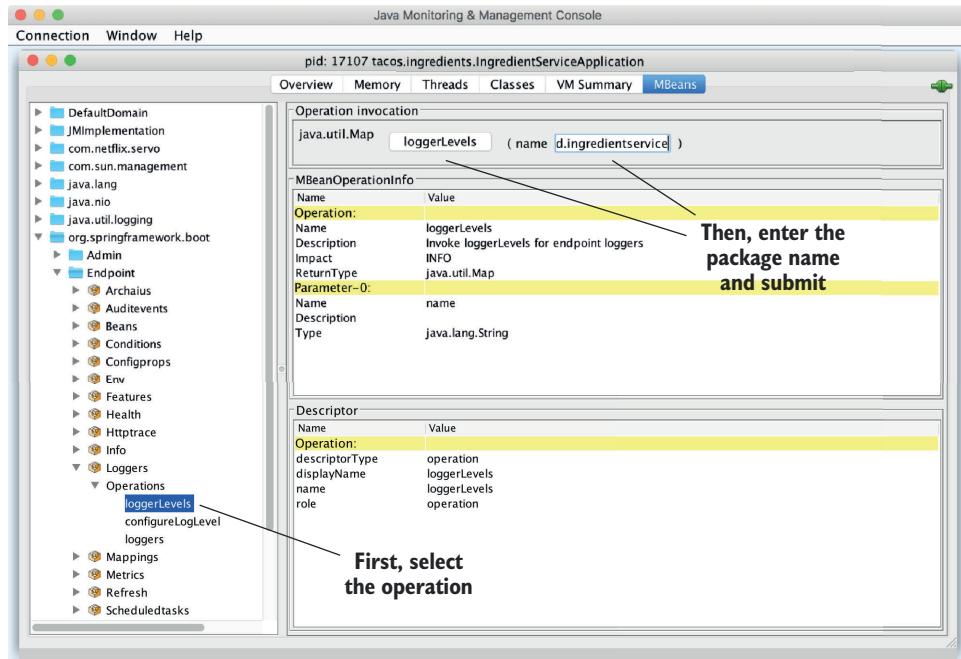


Figure 18.2 Using JConsole to display logging levels from a Spring Boot application

After you click the `loggerLevels` button, a dialog box will pop up showing you the response from the `/loggers` endpoint MBean. It might look a little like figure 18.3.

Although the JConsole UI is a bit clumsy to work with, you should be able to get the hang of it and use it to explore any Actuator endpoint in much the same way.



Figure 18.3 Logging levels from the /loggers endpoint MBean displayed in JConsole

If you don't like JConsole, that's fine—there are plenty of other JMX clients to choose from.

18.2 Creating your own MBeans

Spring makes it easy to expose any bean you want as a JMX MBean. All you must do is annotate the bean class with `@ManagedResource` and then annotate any methods or properties with `@ManagedOperation` or `@ManagedAttribute`. Spring will take care of the rest.

For example, suppose you want to provide an MBean that tracks how many tacos have been ordered through Taco Cloud. You can define a service bean that keeps a running count of how many tacos have been created. The following listing shows what such a service might look like.

Listing 18.1 An MBean that counts how many tacos have been created

```
package tacos.tacos;
import java.util.concurrent.atomic.AtomicLong;
import org.springframework.data.rest.core.event.AbstractRepositoryEventListener;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Service;

@Service
@ManagedResource
public class TacoCounter
    extends AbstractRepositoryEventListener<Taco> {

    private AtomicLong counter;

    public TacoCounter(TacoRepository tacoRepo) {
        long initialCount = tacoRepo.count();
        this.counter = new AtomicLong(initialCount);
    }

    @Override
    protected void onAfterCreate(Taco entity) {
        counter.incrementAndGet();
    }
}
```

```

@ManagedAttribute
public long getTacoCount() {
    return counter.get();
}

@ManagedOperation
public long increment(long delta) {
    return counter.addAndGet(delta);
}
}

```

The TacoCounter class is annotated with `@Service` so that it will be picked up by component scanning, and an instance will be registered as a bean in the Spring application context. But it's also annotated with `@ManagedResource` to indicate that this bean should also be an MBean. As an MBean, it will expose one attribute and one operation. The `getTacoCount()` method is annotated with `@ManagedAttribute` so that it will be exposed as an MBean attribute, while the `increment()` method is annotated with `@ManagedOperation`, exposing it as an MBean operation.

Figure 18.4 shows how the TacoCounter MBean appears in JConsole.

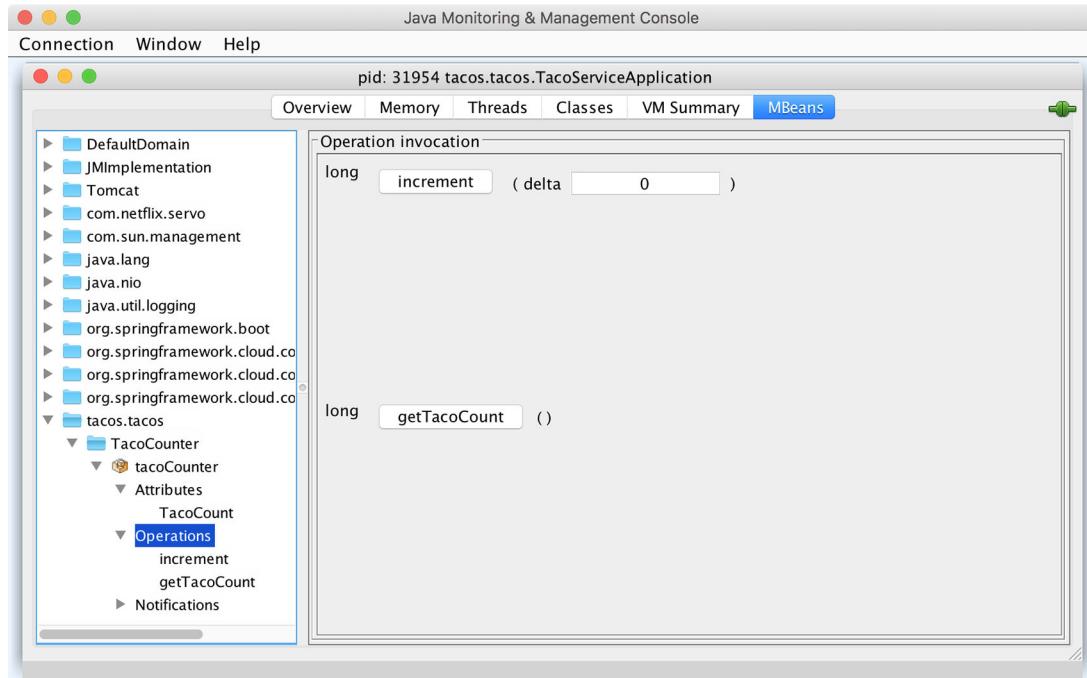


Figure 18.4 TacoCounter's operations and attributes as seen in JConsole

TacoCounter has another trick up its sleeve, although it has nothing to do with JMX. Because it extends `AbstractRepositoryEventListener`, it will be notified of any

persistence events when a Taco is saved through TacoRepository. In this particular case, the `onAfterCreate()` method will be invoked anytime a new Taco object is created and saved, and it will increment the counter by one. But `AbstractRepositoryEventLister` also offers several methods for handling events both before and after objects are created, saved, or deleted.

Working with MBean operations and attributes is largely a pull operation. That is, even if the value of an MBean attribute changes, you won't know until you view the attribute through a JMX client. Let's turn the tables and see how you can push notifications from an MBean to a JMX client.

18.3 **Sending notifications**

MBeans can push notifications to interested JMX clients with Spring's `NotificationPublisher`. `NotificationPublisher` has a single `sendNotification()` method that, when given a `Notification` object, publishes the notification to any JMX clients that have subscribed to the MBean.

For an MBean to be able to publish notifications, it must implement the `NotificationPublisherAware` interface, which requires that a `setNotificationPublisher()` method be implemented. For example, suppose you want to publish a notification every 100 tacos that are created. You can change the `TacoCounter` class so that it implements `NotificationPublisherAware` and uses the injected `NotificationPublisher` to send notifications for every 100 tacos that are created. The following listing shows the changes that must be made to `TacoCounter` to enable such notifications.

Listing 18.2 Sending notifications for every 100 tacos

```
@Service
@ManagedResource
public class TacoCounter
    extends AbstractRepositoryEventListener<Taco>
    implements NotificationPublisherAware {

    private AtomicLong counter;
    private NotificationPublisher np;

    ...

    @Override
    public void setNotificationPublisher(NotificationPublisher np) {
        this.np = np;
    }

    ...

    @ManagedOperation
    public long increment(long delta) {
        long before = counter.get();
```

```

        long after = counter.addAndGet(delta);
        if ((after / 100) > (before / 100)) {
            Notification notification = new Notification(
                "taco.count", this,
                before, after + "th taco created!");
            np.sendNotification(notification);
        }

        return after;
    }
}

```

In the JMX client, you'll need to subscribe to the TacoCounter MBean to receive notifications. Then, as tacos are created, the client will receive notifications for each century count. Figure 18.5 shows how the notifications may appear in JConsole.

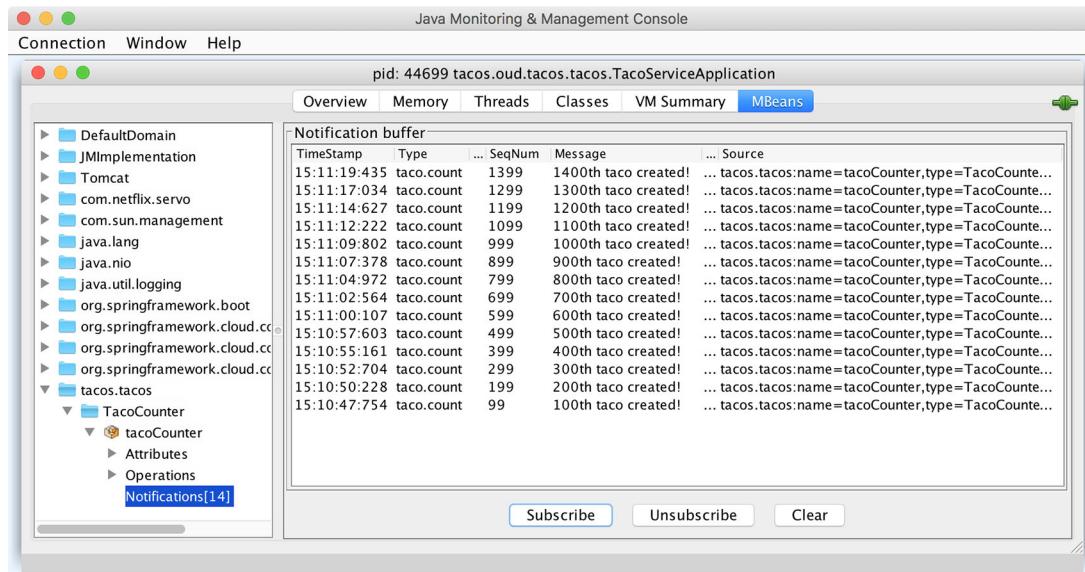


Figure 18.5 JConsole, subscribed to the TacoCounter MBean, receives a notification for every 100 tacos that are created.

Notifications are a great way for an application to actively send data and alerts to a monitoring client without requiring the client to poll managed attributes or invoke managed operations.

Summary

- Most Actuator endpoints are available as MBeans that can be consumed using any JMX client.
- Spring automatically enables JMX for monitoring beans in the Spring application context.
- Spring beans can be exposed as MBeans by annotating them with `@ManagedResource`. Their methods and properties can be exposed as managed operations and attributes by annotating the bean class with `@ManagedOperation` and `@ManagedAttribute`.
- Spring beans can publish notifications to JMX clients using `NotificationPublisher`.

19

Deploying Spring

This chapter covers

- Building Spring applications as either WAR or JAR files
- Pushing Spring applications to Cloud Foundry
- Containerizing Spring applications with Docker

Think of your favorite action movie. Now imagine going to see that movie in the theater and being taken on a thrilling audiovisual ride with high-speed chases, explosions, and battles, only to have it come to a sudden halt before the good guys take down the bad guys. Instead of seeing the movie's conflict resolved, when the theater lights come on, everyone is ushered out the door. Although the lead-up was exciting, it's the climax of the movie that's important. Without it, it's action for action's sake.

Now imagine developing applications and putting a lot of effort and creativity into solving the business problem, but then never deploying the application for others to use and enjoy. Sure, most applications we write don't involve car chases or explosions (at least I hope not), but there's a certain rush you get along the way. Not every line of code you write is destined for production, but it'd be a big let-down if none of it ever was deployed.

Up to this point, we've focused on using the features of Spring Boot that help us develop an application. There have been some exciting steps along the way. But it's all for nothing if you don't cross the finish line and deploy the application.

In this chapter, we're going to step beyond developing applications with Spring Boot and look at how to deploy those applications. Although this may seem obvious for anyone who has ever deployed a Java-based application, there are some unique features of Spring Boot and related Spring projects you can draw on that make deploying Spring Boot applications unique.

In fact, unlike most Java web applications, which are typically deployed to an application server as WAR files, Spring Boot offers several deployment options. Before we look at how to deploy a Spring Boot application, let's consider all the options and choose a few that suit your needs best.

19.1 Weighing deployment options

You can build and run Spring Boot applications in several ways. The appendix covers many of them, including these:

- Running the application in the IDE with either Spring Tool Suite or IntelliJ IDEA
- Running the application from the command line using the Maven `spring-boot:run` goal or Gradle `bootRun` task
- Using Maven or Gradle to produce an executable JAR file that can be run at the command line or deployed in the cloud
- Using Maven or Gradle to produce a WAR file that can be deployed to a traditional Java application server

Any of these choices is suitable for running the application while you're still developing it. But what about when you're ready to deploy the application into a production or other non-development environment?

Although running an application from the IDE or via Maven or Gradle aren't considered production-ready options, executable JAR files and traditional Java WAR files are certainly valid options for deploying applications to a production environment. Given the options of deploying a WAR file or a JAR file, how do you choose? In general, the choice comes down to whether you plan to deploy your application to a traditional Java application server or to a cloud platform:

- *Deploying to Java application servers*—If you must deploy your application to Tomcat, WebSphere, WebLogic, or any other traditional Java application server, you really have no choice but to build your application as a WAR file.
- *Deploying to the cloud*—If you're planning to deploy your application to the cloud, whether it be Cloud Foundry, Amazon Web Services (AWS), Azure, Google Cloud Platform, or most any other cloud platform, then an executable JAR file is the best choice. Even if the cloud platform supports WAR deployment, the JAR file format is much simpler than the WAR format, which is designed for application server deployment.

In this chapter, we'll focus on three deployment scenarios:

- Deploying a Spring Boot application as a WAR file to a Java application server such as Tomcat
- Pushing a Spring Boot application as an executable JAR file to Cloud Foundry
- Packaging a Spring Boot application into a Docker container for deployment to any platform that supports Docker deployments

To get started, let's take a look at how you can build the ingredient service application into a WAR file that can be deployed to a Java application server such as Tomcat.

19.2 Building and deploying WAR files

Throughout the course of this book, as you've developed the applications that make up the Taco Cloud application, you've run them either in the IDE or from the command line as an executable JAR file. In either case, an embedded Tomcat server (or Netty, in the case of Spring WebFlux applications) has always been there to serve requests to the application.

Thanks in large part to Spring Boot autoconfiguration, you've been spared from having to create a web.xml file or servlet initializer class to declare Spring's DispatcherServlet for Spring MVC. But if you're going to deploy the application to a Java application server, you're going to need to build a WAR file. And, so that the application server will know how to run the application, you'll also need to include a servlet initializer in that WAR file to play the part of a web.xml file and declare DispatcherServlet.

As it turns out, building a Spring Boot application into a WAR file isn't all that difficult. In fact, if you chose the WAR option when creating the application through the Initializr, then there's nothing more you need to do.

The Initializr ensures that the generated project will contain a servlet initializer class and the build file will be geared to produce a WAR file. If, however, you chose to build a JAR file from the Initializr (or if you're curious as to what the pertinent differences are), then read on.

First, you'll need a way to configure Spring's DispatcherServlet. Whereas this could be done with a web.xml file, Spring Boot makes this even easier with `SpringBootServletInitializer`. `SpringBootServletInitializer` is a special Spring Boot-aware implementation of Spring's `WebApplicationInitializer`. Aside from configuring Spring's DispatcherServlet, `SpringBootServletInitializer` also looks for any beans in the Spring application context that are of type `Filter`, `Servlet`, or `ServletContextInitializer` and binds them to the servlet container.

To use `SpringBootServletInitializer`, create a subclass and override the `configure()` method to specify the Spring configuration class. Listing 19.1 shows `IngredientServiceServletInitializer`, a subclass of `SpringBootServletInitializer` that you'll use for the ingredient service application.

Listing 19.1 Enabling Spring web applications via Java

```
package tacos.ingredients;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.web.SpringBootServletInitializer;

public class IngredientServiceServletInitializer
    extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder builder) {
        return builder.sources(IngredientServiceApplication.class);
    }
}
```

As you can see, the `configure()` method is given a `SpringApplicationBuilder` as a parameter and returns it as a result. In between, it calls the `sources()` method that registers Spring configuration classes. In this case, it only registers the `IngredientServiceApplication` class, which serves the dual purpose of a bootstrap class (for executable JARs) and a Spring configuration class.

Even though the ingredient service application has other Spring configuration classes, it's not necessary to register them all with the `sources()` method. The `IngredientServiceApplication` class, annotated with `@SpringBootApplication`, implicitly enables component scanning. Component scanning discovers and pulls in any other configuration classes that it finds.

For the most part, `SpringBootServletInitializer`'s subclass is boilerplate. It references the application main configuration class. But aside from that, it'll be the same for every application where you'll be building a WAR file. And you'll almost never need to make any changes to it.

Now that you've written a servlet initializer class, you must make a few small changes to the project build. If you're building with Maven, the change required is as simple as ensuring that the `<packaging>` element in `pom.xml` is set to `war`:

```
<packaging>war</packaging>
```

The changes required for a Gradle build are similarly straightforward. You must apply the `war` plugin in the `build.gradle` file:

```
apply plugin: 'war'
```

Now you're ready to build the application. With Maven, you'll use the Maven wrapper script that the Initializr used to execute the package goal:

```
$ mvnw package
```

If the build is successful, then the WAR file can be found in the target directory. On the other hand, if you were using Gradle to build the project, you'd use the Gradle wrapper to execute the build task:

```
$ gradlew build
```

Once the build completes, the WAR file will be in the build/libs directory. All that's left is to deploy the application. The deployment procedure varies across application servers, so consult the documentation for your application server's specific deployment procedure.

It may be interesting to note that although you've built a WAR file suitable for deployment to any Servlet 3.0 (or higher) servlet container, the WAR file can still be executed at the command line as if it were an executable JAR file:

```
$ java -jar target/ingredient-service-0.0.19-SNAPSHOT.war
```

In effect, you get two deployment options out of a single deployment artifact!

MICROSERVICES IN APPLICATION SERVERS?

The ingredient service application is intended to be one of several applications that are microservice constituents of the larger Taco Cloud application. But here, we're talking about deploying the ingredient service as a standalone application to an application server. Does that even make sense?

Microservices are generally like any other application and should be deployable on their own. Although the ingredient service may not be useful outside the context of the rest of the Taco Cloud application, there's no reason you can't deploy it to Tomcat or another application server. But don't expect the same ability to scale the application individually as you would get if deploying it to the cloud.

Although WAR files have been the workhorses of Java deployment for over 20 years, they were truly designed for deploying applications to a traditional Java application server. Depending on the platform you choose, modern cloud deployment doesn't require WAR files and some may not even support them. As we move into a new era of cloud deployment, perhaps JAR files are a better choice.

19.3 **Pushing JAR files to Cloud Foundry**

Server hardware can be expensive to purchase and maintain. Properly scaling servers to handle heavy loads can be tricky and even prohibitive for some organizations. These days, deploying applications to the cloud is a compelling and cost-effective alternative to running your own data center.

Several cloud choices are available, but those that offer a platform as a service (PaaS) are among the most compelling. PaaS offers a ready-made application deployment platform with several add-on services (such as databases and message brokers) to bind to your applications. In addition, as your application requires additional

horsepower, cloud platforms make it easy to scale up (or down) your application on the fly by adding and removing instances.

Cloud Foundry is an open source PaaS platform that originated at Pivotal, the same company that sponsors the Spring Framework and the other libraries in the Spring platform. One of the most compelling things about Cloud Foundry is that it offers both open source and commercial-based distributions, giving you the choice of how and where you use Cloud Foundry. It can even be run inside the firewall in a corporate data center, offering a private cloud.

Whereas Cloud Foundry will be happy to accept WAR files, the WAR file format is overkill for Cloud Foundry's needs. A simpler executable JAR file is a more suitable choice for deploying to Cloud Foundry.

To demonstrate how to build and deploy an executable JAR file to Cloud Foundry, you're going to build the ingredient service application and deploy it to Pivotal Web Services (PWS), a public Cloud Foundry hosted by Pivotal at <http://run.pivot.io>. If you want to work with PWS, you'll need to sign up for an account. PWS offers \$87 of free trial credit and doesn't even require you to give any credit card information during the trial.

Once you've signed up for PWS, you'll need to download and install the cf command-line tool from <https://console.run.pivot.io/tools>. You'll use the cf tool to push applications to Cloud Foundry. But the first thing you'll use it for is to log into your PWS account:

```
$ cf login -a https://api.run.pivot.io  
API endpoint: https://api.run.pivot.io
```

```
Email> {your email}
```

```
Password> {your password}
```

```
Authenticating...
```

```
OK
```

Great! Now you're ready to take the ingredient service to the cloud! As it turns out, the project is ready to be deployed to Cloud Foundry. All you need to do is build it and then push it to the cloud.

To build the project with Maven, you can use the Maven wrapper to execute the package goal (you'll find the resulting JAR file in the target directory):

```
$ mvnw package
```

With Gradle, you can use the Gradle wrapper to execute the build task (you'll find the resulting JAR file in the build/libs directory):

```
$ gradlew build
```

Now all that's left is to push the JAR file to Cloud Foundry using the cf command:

```
$ cf push ingredient-service -p target/ingredient-service-0.0.19-SNAPSHOT.jar
```

The first argument to `cf push` is the name given to the application in Cloud Foundry. In this case, the full URL for the application will be <http://ingredient-service.cfapps.io>. Among other things, this name will be used as the subdomain where the application is hosted. Therefore, it's important that the name you give the application be unique so that it doesn't collide with any other applications deployed in Cloud Foundry (including those deployed by other Cloud Foundry users).

Because dreaming up a unique name can be tricky, the `cf push` command offers a `--random-route` option that randomly produces a subdomain for you. Here's how to push the ingredient service application to generate a random route:

```
$ cf push ingredient-service \
  -p target/ingredient-service-0.0.19-SNAPSHOT.jar \
  --random-route
```

When using `--random-route`, the application name is still required, but two randomly chosen words will be appended to it to produce the subdomain.

Assuming everything goes well, the application should be deployed and ready to handle requests. Supposing that the subdomain is `ingredient-service`, you can point your browser to <http://ingredient-service.cfapps.io/ingredients> to see it in action. You should receive, as a response, a list of available ingredients.

As written, the application will continue to use the embedded Mongo database (which is only intended for testing purposes) to hold ingredient data. You'll likely want to use a real database in production. At the time I'm writing this, there's a fully managed MongoDB service available in PWS under the name `mlab`. You can find this service (and any other available services) by using the `cf marketplace` command. To create an instance of the `mlab` service, use the `cf create-service` command:

```
$ cf create-service mlab sandbox ingredientdb
```

This creates an `mlab` service with the `sandbox` service plan named `ingredientdb`. Once the service is created, you can bind it to your application with the `cf bind-service` command. For example, to bind the `ingredientdb` service to the `ingredient service` application, use this:

```
$ cf bind-service ingredient-service ingredientdb
```

Binding a service to an application merely provides the application with details on how to connect to the service with an environment variable named `VCAP_SERVICES`. It doesn't change the application in any way to use that service. Once the service is bound, you'll need to re-stage the application to have the binding take effect:

```
$ cf restage ingredient-service
```

The `cf restage` command forces Cloud Foundry to redeploy the application and reevaluate the `VCAP_SERVICES` value. When it does, it'll see that there's a MongoDB

service bound to the application. It uses that service as the backing database for the application.

There are dozens of available services in PWS that you can bind your application to, including MySQL databases, PostgreSQL databases, and even ready-to-use Eureka and Config Server services. I encourage you to read more about what PWS has to offer at <https://console.run.pivotal.io/marketplace> and acquaint yourself with how to use PWS by reading the documentation at <https://docs.run.pivotal.io/>.

Cloud Foundry is a great PaaS for Spring Boot application deployment. Its association with the Spring projects affords some synergy between the two. But another common way to deploy applications in the cloud, especially when pushing to an Infrastructure-as-a-Service (IAAS) platform like AWS, is to package the application within a Docker container that's published to the cloud. Let's see how to create a Docker container that carries your Spring Boot application.

19.4 Running Spring Boot in a Docker container

Docker (<https://www.docker.com/>) has become the de facto standard for distributing applications of all kinds for deployment in the cloud. Many different cloud environments, including AWS, Microsoft Azure, Google Cloud Platform, and Pivotal Web Services (to name a few) accept Docker containers for deploying applications.

The idea of containerized applications, such as those created with Docker, draws analogies from real-world intermodal containers. With regard to shipping items, intermodal containers all have a standard size and format, regardless of their contents. Because of that, intermodal containers are easily stacked on ships, carried on trains, or pulled by trucks. In a similar way, containerized applications share a common container format that can be deployed and run anywhere, regardless of the application inside.

Although creating Docker images isn't terribly difficult, Spotify has created a Maven plugin that makes creating a Docker container from the result of a Spring Boot build as easy as whistling your favorite tune. To use the Docker plugin, add it to your Spring Boot project pom.xml file under the `<build>/<plugins>` block as follows:

```
<build>
  <plugins>
  ...
    <plugin>
      <groupId>com.spotify</groupId>
      <artifactId>dockerfile-maven-plugin</artifactId>
      <version>1.4.3</version>
      <configuration>
        <repository>
          ${docker.image.prefix}/${project.artifactId}
        </repository>
        <buildArgs>
          <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
        </buildArgs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```

</configuration>
</plugin>
</plugins>
</build>

```

Under the `<configuration>` block, you'll set a few properties to guide the creation of the Docker image. The `<repository>` element describes the name of the Docker image as it'll appear in a Docker repository. As specified here, the name is based on the Maven project artifact ID, prefixed with a value resolved from the Maven property named `docker.image.prefix`. Although the project artifact ID is something Maven already knows, you'll need to specify the `prefix` property:

```

<properties>
  ...
  <docker.image.prefix>tacocloud</docker.image.prefix>
</properties>

```

If this were the Taco Cloud ingredient service, the resulting Docker image would reside in the Docker repository as `tacocloud/ingredient-service`.

Under the `<buildArgs>` element, you can guide the image to include the JAR file that the Maven build produces. As shown, it uses the Maven property `project.build.finalName` to determine the name of the JAR file that's in the target directory.

Aside from the information you provided in the Maven build specification, all Docker images are defined from a file named Dockerfile. This file identifies an image to base the new image on, environment variables that should be set, any volumes that should be mounted, and (most importantly) the *entry point*—a command to execute when a container based on the image starts. For the purposes of most any Spring Boot application, the following Dockerfile is a great way to begin:

```

FROM openjdk:8-jdk-alpine
ENV SPRING_PROFILES_ACTIVE docker
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", \
            "-Djava.security.egd=file:/dev/./urandom", \
            "-jar", \
            "/app.jar"]

```

Breaking this Docker file down line by line, you see the following:

- The `FROM` instruction identifies an image to base the new image on. The new image extends the base image. In this case, the base image is `openjdk:8-jdk-alpine`, a container image based on version 8 of OpenJDK.
- The `ENV` instruction sets an environment variable. You're going to override a few of the Spring Boot application configuration properties based on the active profile, so in this image, you'll set the environment variable `SPRING_PROFILES_ACTIVE`

to docker to ensure that the Spring Boot application starts with docker as the active profile.

- The `VOLUME` instruction creates a mount point in the container. In this case, it creates a mount point at `/tmp` so that the container can write data, if necessary, to the `/tmp` directory.
- The `ARG` instruction declares an argument that can be passed in at build time. In this case, it declares an argument named `JAR_FILE`, which is the same as the argument given in the Maven plugin's `<buildArgs>` block.
- The `COPY` instruction copies a file from a given path to another path. In this case, it copies the JAR file specified in the Maven plugin to a file named `app.jar` within the image.
- The `ENTRYPOINT` instruction describes what should happen when the container starts. Given as an array, it specifies the command line to execute. In this case, it uses the `java` command line to run the executable `app.jar`.

Draw special attention to the `ENV` instruction. It's generally a good idea to set the `SPRING_PROFILES_ACTIVE` environment variable in any container image that contains a Spring Boot application. This makes it possible to configure beans and configuration properties that are unique to applications running in Docker.

In the case of the ingredient service, you're going to need a way to link the application to a Mongo database running in a separate container. By default, Spring Data attempts to connect to a Mongo database listening at port 27017 on localhost. But that was only the case when running everything locally and not in any containers. You'll need to configure the `spring.data.mongodb.host` property to tell Spring Data the hostname where Mongo will be available.

Although you may not yet know where the Mongo database will be running, you can configure Spring Data to connect to Mongo on a host named `mongo` when the `docker` profile is active by adding the following Docker-specific configuration to the `application.yml` file:

```
---
spring:
  profiles: docker

data:
  mongodb:
    host: mongo
```

In a moment, when you fire up the Docker container, you'll map the `mongo` host to a Mongo database running in a different container. But now you're ready to build the Docker image. Using the Maven wrapper, execute the `package` and `dockerfile:build` goals to build the JAR file, and then build the Docker image:

```
$ mvnw package dockerfile:build
```

At this point, you can verify that the image is in your local image repository by using the docker images command (the CREATED and SIZE columns were omitted for easier readability and to fit within the margins of this page):

```
$ docker images
REPOSITORY           TAG      IMAGE ID
tacocloud/ingredient-service  latest   7e8ed20e768e
```

Before you can start the container, you'll need to start a container for the Mongo database. The following command line runs a new Docker container named tacocloud-mongo with a Mongo 3.7.9 database:

```
$ docker run --name tacocloud-mongo -d mongo:3.7.9-xenial
```

Now, you can finally run the ingredient service container, linking it to the Mongo container you just started:

```
$ docker run -p 8080:8081 \
             --link tacocloud-mongo:mongo \
             tacocloud/ingredient-service
```

The docker run command shown here has several important components worth noting:

- Because you've configured the Spring Boot application in the container to run on port 8081, the -p parameter maps the internal port to the host's port 8080.
- The --link parameter links your container to the container named tacocloud-mongo and assigns it a hostname of mongo so that Spring Data can connect to it with that hostname.
- Finally, you specify the name of the image (in this case, tacocloud/ingredient-service) to run in a new container.

Now that you have a Docker image built and have proven it to run as a local container, you can take it to the next level by pushing the image to Dockerhub or some other Docker image repository. If you have an account on Dockerhub and are logged in, then you can push the image using Maven like this:

```
$ mvnw dockerfile:push
```

From that point, you can deploy the image to almost any environment that supports Docker containers, including AWS, Microsoft Azure, and Google Cloud Platform. Pick your environment and follow the platform-specific instructions for deploying Docker containers. Here are links to instructions for a few popular cloud platforms:

- *AWS*—<https://aws.amazon.com/getting-started/tutorials/deploy-docker-containers/>
- *Microsoft Azure*—<https://docs.docker.com/docker-for-azure/deploy/>
- *Google Cloud Platform*—<https://cloud.google.com/kubernetes-engine/docs/tutorials/hello-app>

- *Pivotal Web Services (PWS)*—<https://docs.run.pivotal.io/devguide/deploy-apps/push-docker.html>
- *Pivotal Container Service (PKS)*—<https://pivotal.io/platform/pivotal-container-service>

19.5 The end is where we begin

Over the past few hundred pages, we've gone from a simple start—or `start.spring.io`, more specifically—to deploying an application in the cloud. I hope that you've had as much fun working through these pages as I've had writing them.

But while this book must come to an end, your Spring adventure is just beginning. Using what you've learned in these pages, go build something amazing with Spring. I can't wait to see what you come up with!

Summary

- Spring applications can be deployed in a number of different environments, including traditional application servers, platform-as-a-service (PaaS) environments like Cloud Foundry, or as Docker containers.
- When building a WAR file, you should include a class that subclasses `SpringBootServletInitializer` to ensure that Spring's `DispatcherServlet` is properly configured.
- Building as an executable JAR file allows a Spring Boot application to be deployed to several cloud platforms without the overhead of a WAR file.
- Containerizing Spring applications is as simple as using Spotify's Dockerfile plugin for Maven. It wraps an executable JAR file in a Docker container that can be deployed anywhere Docker containers can be deployed, including cloud providers such as Amazon Web Services, Microsoft Azure, Google Cloud Platform, Pivotal Web Services (PWS), and Pivotal Container Service (PKS).

appendix *Bootstrapping* *Spring applications*

There are a lot of ways to kick-start your Spring projects, and which you choose is largely a matter of personal taste. Many of the choices will be decided by which is your favorite IDE.

All but one of these options are based on the Spring Initializr, which is a REST API that generates Spring Boot projects for you. The various IDE choices are nothing more than clients for that REST API. Additionally, there are a few ways to use the Spring Initializr API outside of your IDE.

This appendix takes a quick look at all of these options.

A.1 *Initializing a project with Spring Tool Suite*

To initialize a new Spring project with Spring Tool Suite, choose the Spring Starter Project menu option from the File > New menu, as shown in figure A.1.

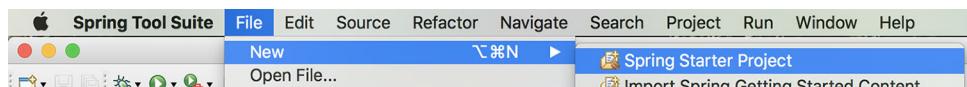


Figure A.1 Starting a new project in Spring Tool Suite

NOTE This is an abbreviated description of using Spring Tool Suite to initialize a Spring project. For a more detailed explanation, see section 1.2.1.

You'll be shown the first page of the project creation dialog box (figure A.2). On this page, you'll define basic project information, such as the project's name, coordinates (group ID and artifact ID), version, and base package name. You can also specify whether the project will be built with Maven or Gradle, whether the build

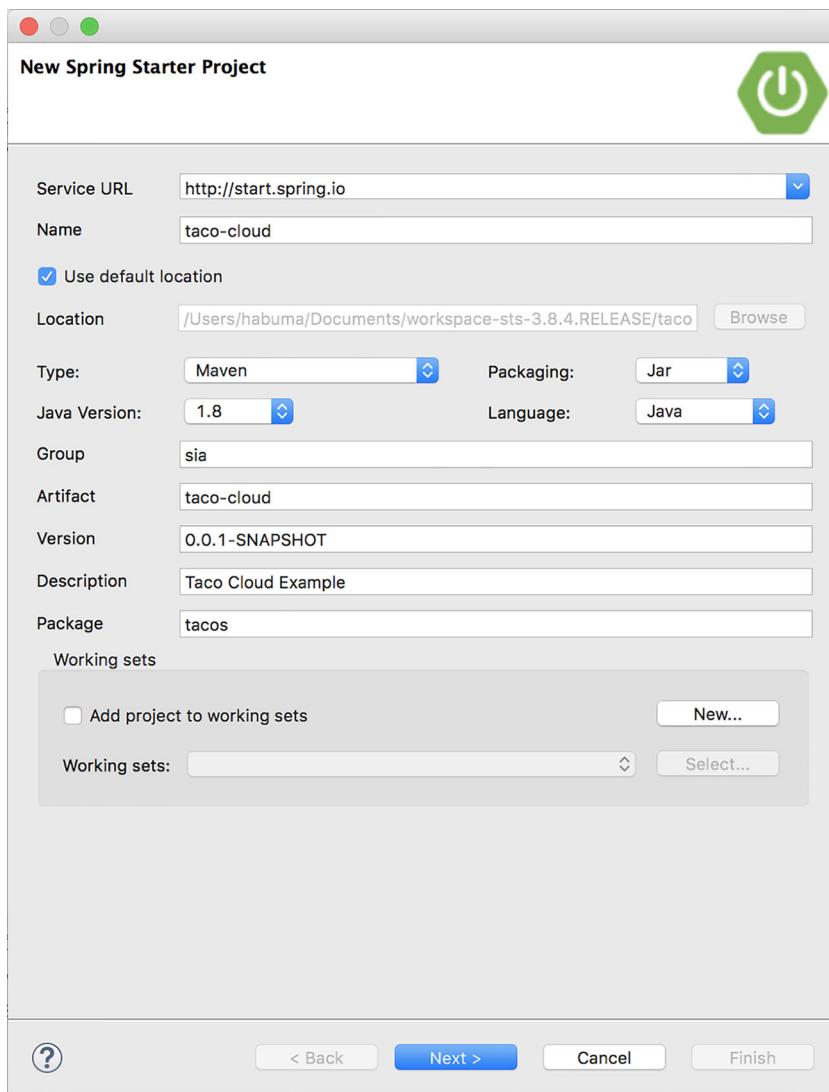


Figure A.2 Defining basic project information

will produce a JAR file or a WAR file, which version of Java to build with, and even an alternate JVM language to use, such as Groovy or Kotlin.

The first field on this page asks you to specify the location of the Spring Initializr service. If you're running or using a custom instance of the Initializr, you'll want to specify the base URL of the Initializr service here. Otherwise, you'll be fine leaving it with the default that points to <http://start.spring.io>.

After you've defined the basic project information, click Next to see the project dependencies page (figure A.3).

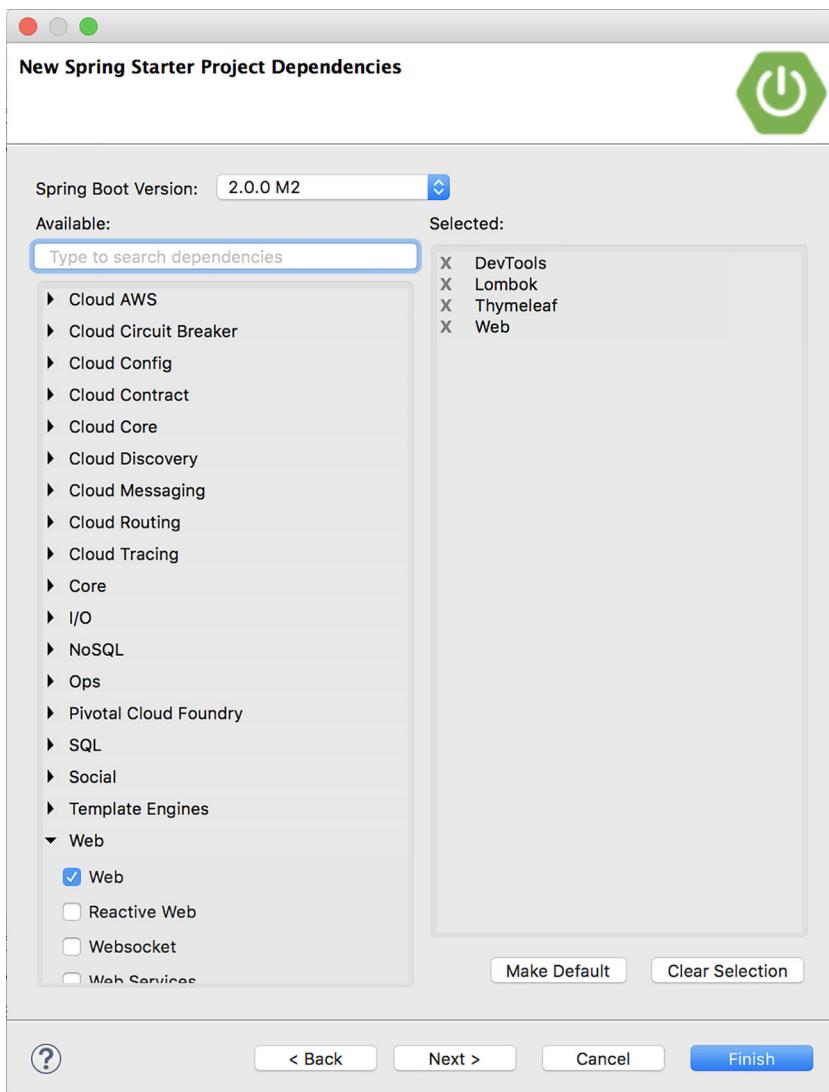


Figure A.3 Specifying project dependencies

On the project dependencies page, you can specify all of the dependencies your project will need. Many of these dependencies are Spring Boot Starter dependencies, although some other dependencies are commonly used in Spring projects.

The available dependencies are listed on the left side, organized in groups that can be expanded or collapsed. If you're having trouble finding a dependency, you can also search for dependencies to narrow your choices down.

To add a dependency to the generated project, check the check box next to the dependency name. Your selections will appear in the list on the right side under

the Selected header. You can remove a dependency by clicking the X next to the selected dependency. Or click Clear Selection to remove all selected dependencies.

As an added convenience, if you find that you have a certain core set of dependencies that you always (or often) use for your projects, you can click the Make Default button after selecting those dependencies, and they'll already be checked the next time you create a project.

After making your selections, click Finish to generate the project and add it to your workspace.

If, however, you want to use an Initializr other than the one at <http://start.spring.io>, click Next to set the Initializr base URL, as shown in figure A.4.

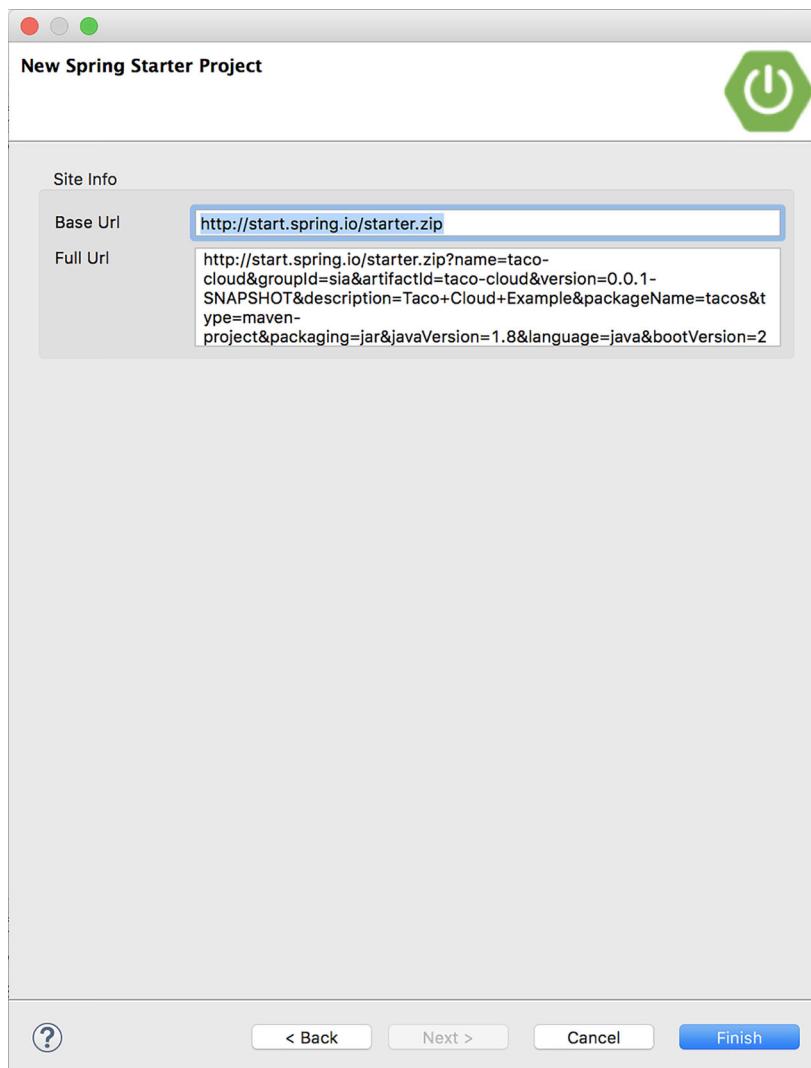


Figure A.4 Optionally specifying the Initializr base URL

The Base Url field specifies the URL where the Initializr API is listening. This is the only field you can change on this page. The Full Url field shows the complete URL that will be used to request a new project from the Initializr.

A.2 **Initializing a project with IntelliJ IDEA**

To get started on a new Spring project in IntelliJ IDEA, choose the Project menu item from the File > New menu, as shown in figure A.5.

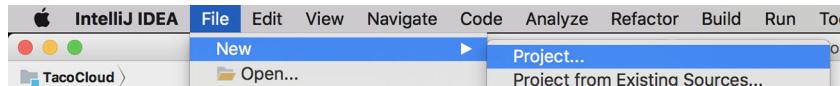


Figure A.5 Starting a new Spring project in IntelliJ IDEA

This opens up the first page of a new Spring Initializr project wizard (see figure A.6). On this page, you can usually just click Next to go to the next page of the wizard. But

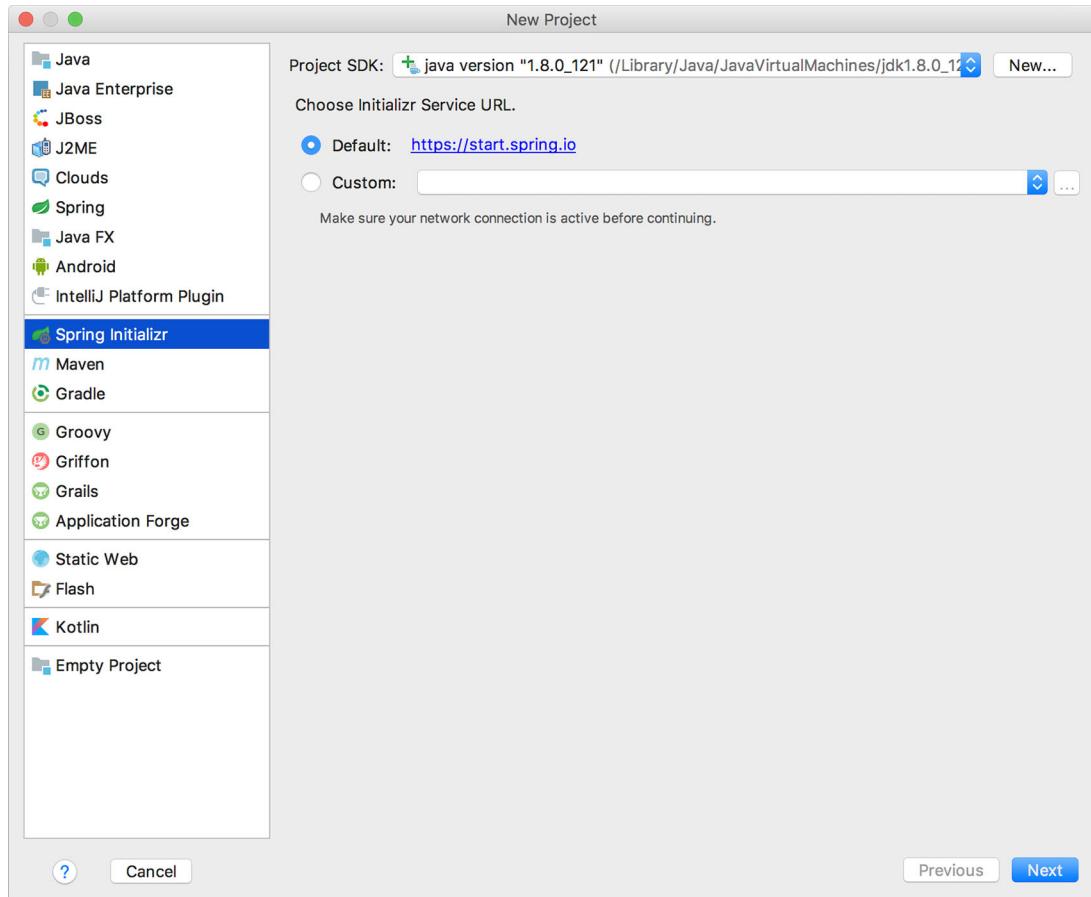


Figure A.6 Selecting the location of the Spring Initializr

if you want to use a Spring Initializr different from the one at <https://start.spring.io>, you'll need to select the Custom radio button and enter the base URL of the Spring Initializr you want to use.

After clicking Next, you'll be presented with a page that asks for essential project information, as shown in figure A.7. You may recognize some of the fields on this page as information that might appear in a Maven pom.xml file—in fact, if you select Maven Project from the Type field, that's exactly how it will be used. You're welcome to choose Gradle Project instead if Gradle is your preference.

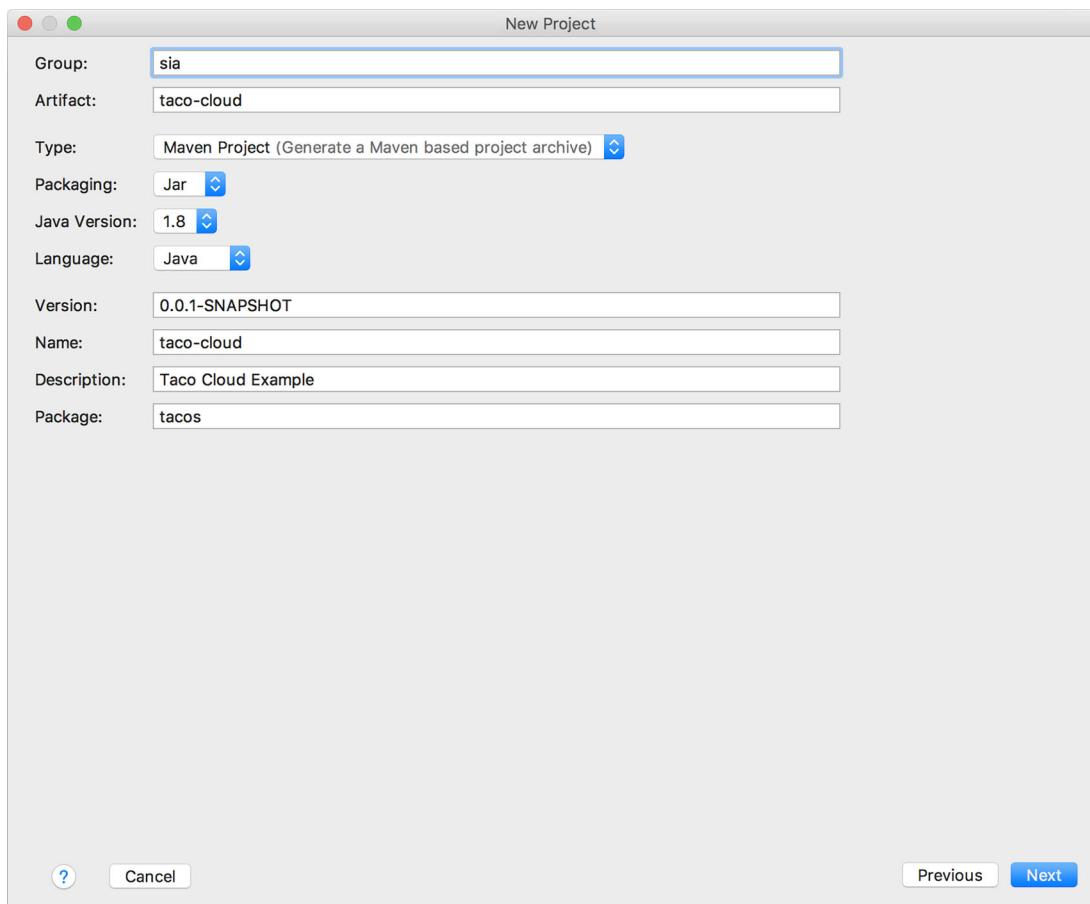


Figure A.7 Specifying essential project information in IntelliJ IDEA

Once you've filled in the essential project information, click Next to be shown the project dependencies page (figure A.8).

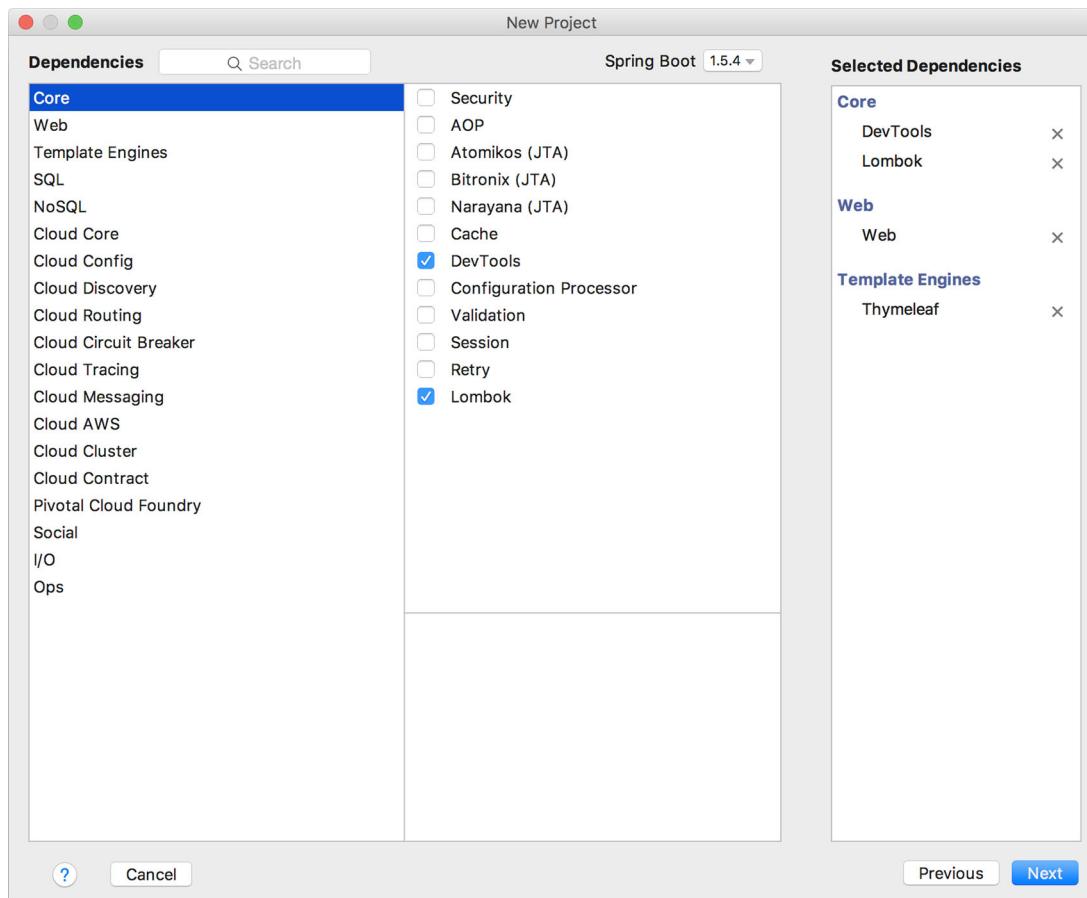


Figure A.8 Selecting project dependencies

The dependencies are organized by category in the far-left list. Selecting a category will result in that category's options being presented in the middle list. Your selected dependencies will be listed (according to category) in the right list.

After all of your dependencies have been selected, click Next. You'll be presented with the final page of the project wizard, as shown in figure A.9, which asks you to name the project and specify where the project should reside on your disk.

Click Finish and your project will be created and loaded into the IntelliJ IDEA workspace.

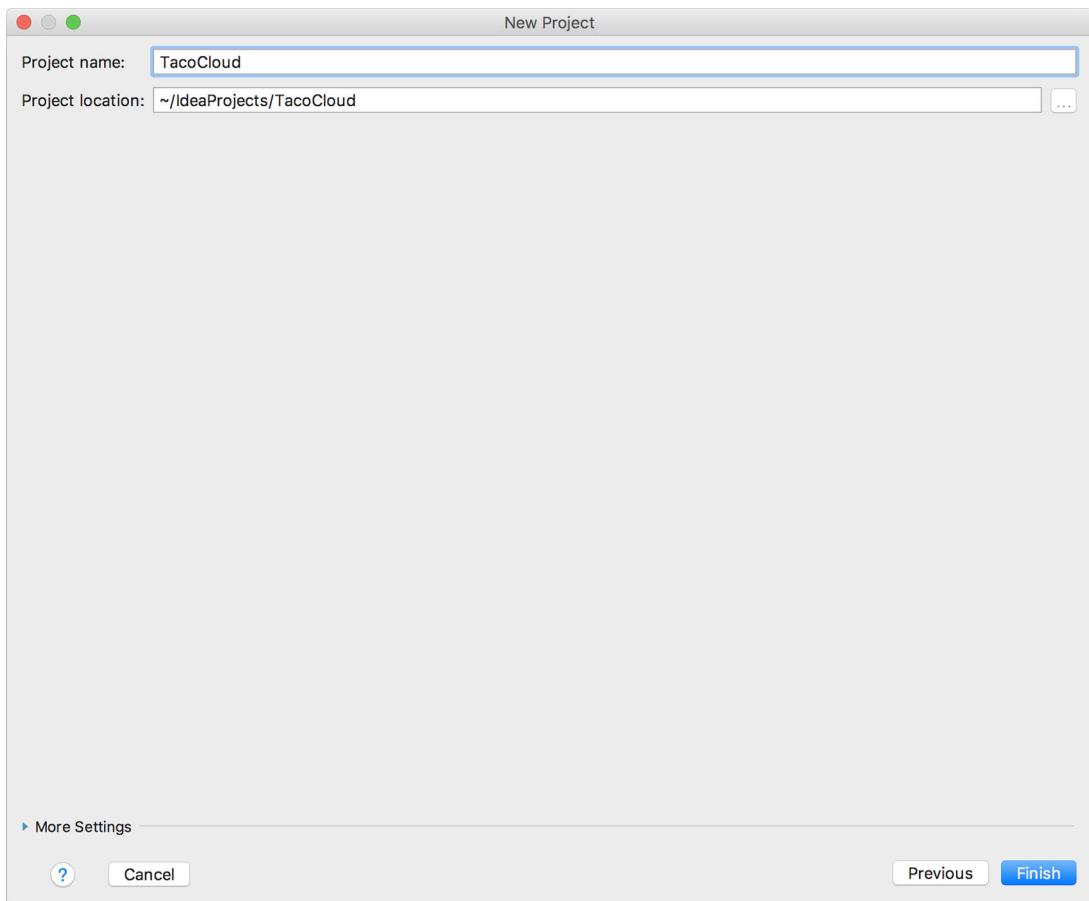


Figure A.9 Setting the project name and location

A.3 **Initializing a project with NetBeans**

To create a new Spring project in NetBeans, start by selecting the New Project menu item under the File menu, as shown in figure A.10.

You'll be shown the first page of the new project wizard. As shown in figure A.11, this page will let you choose what kind of project you want to create.

For a Spring Boot project, select Maven from the category list on the left, and then select Spring Boot Initializr Project from the project list on the right. Then click Next to move to the next page.

The second page in the new project wizard (figure A.12) lets you set essential project information, such as the project name, version, and other information that will ultimately be used to define the project in a Maven pom.xml file.

After you've specified the basic project information, click Next to navigate to the dependencies page in the new project wizard, shown in figure A.13.

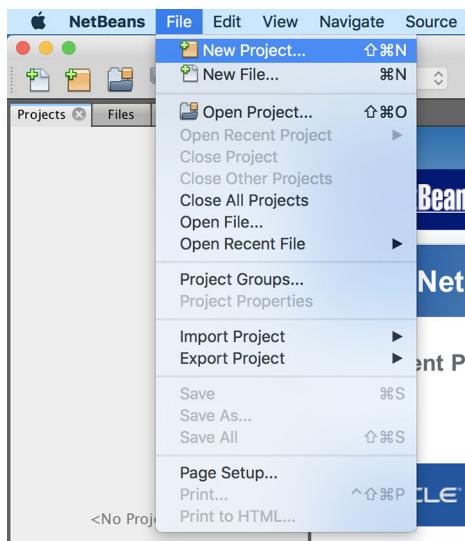


Figure A.10 Starting a new Spring project in NetBeans

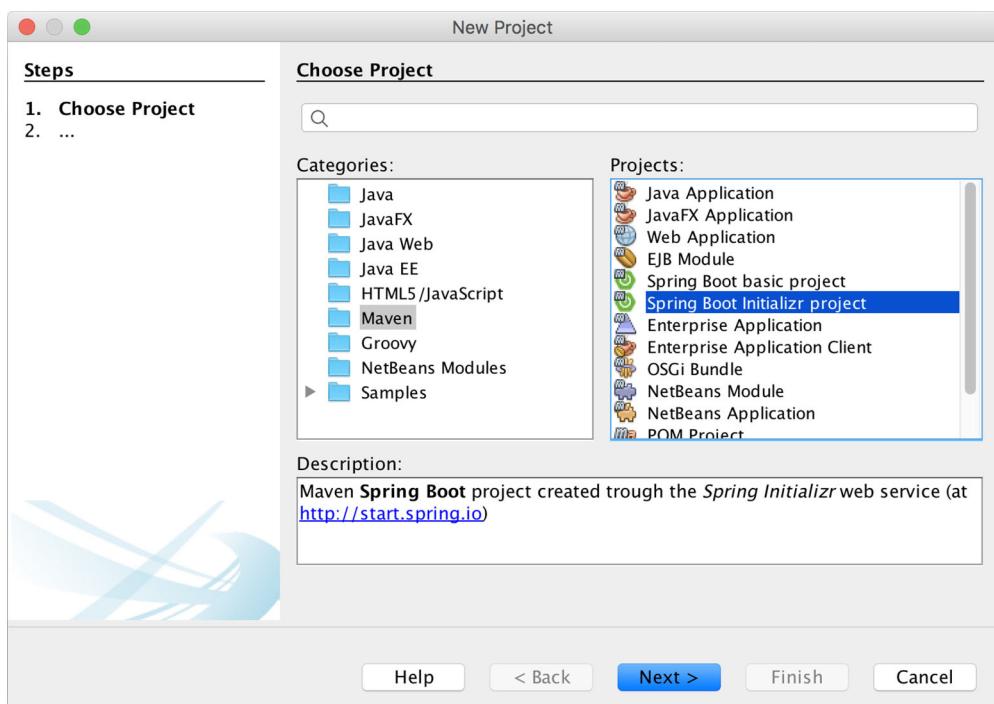


Figure A.11 Creating a new Spring Boot Initializr project

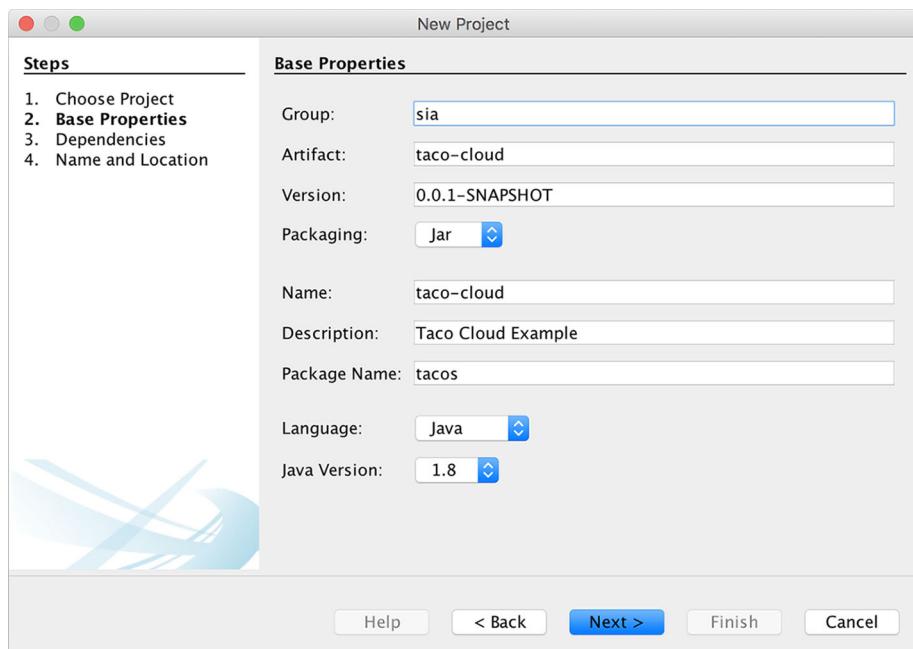


Figure A.12 Specifying essential project information

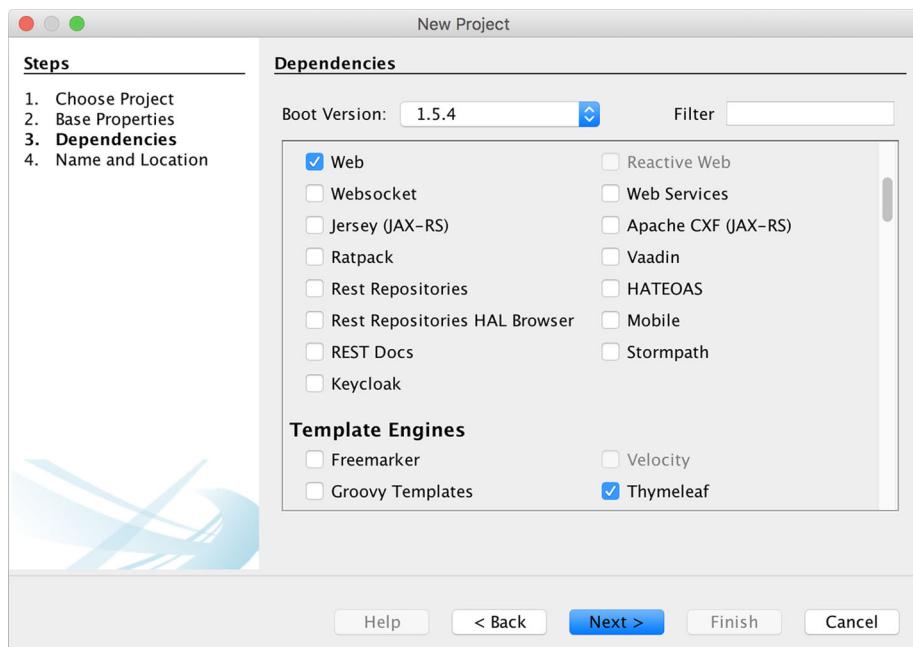


Figure A.13 Selecting project dependencies

Dependencies are all listed as check boxes in the same list, organized by category. If you have trouble finding the specific dependency you need, you can use the Filter text box at the top to limit the list of options.

You can also specify which version of Spring Boot you wish to use on this page. It will be set to the current generally available version of Spring Boot by default.

Once you've selected the dependencies for your project, click Next to navigate to the last page of the new project wizard, shown in figure A.14.

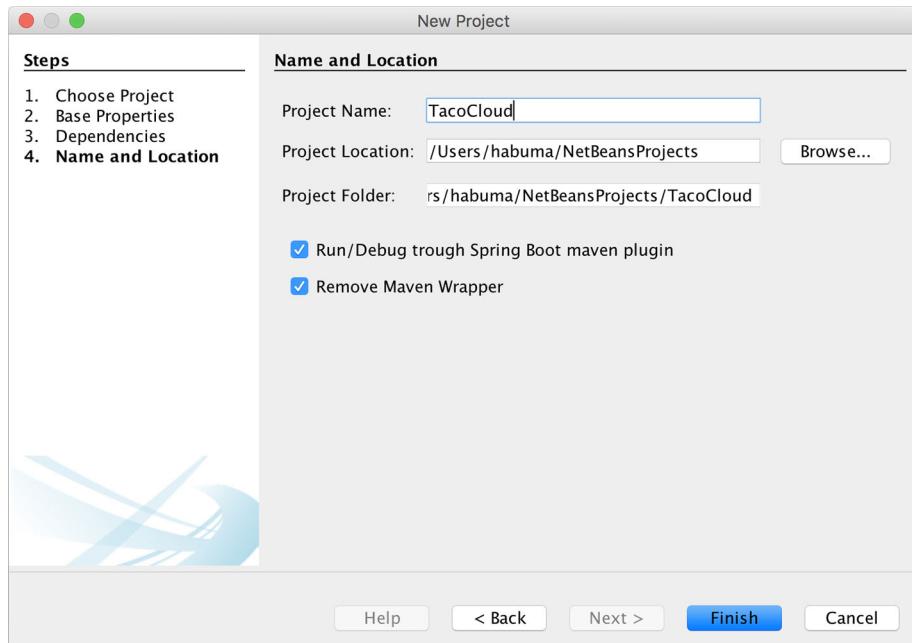


Figure A.14 Specifying the project's name and location

This page lets you specify some final details about the project, including the project name and location on the filesystem. (The Project Folder field is read-only and derived from the other two fields.) It also gives you the option to run and debug your project through the Maven Spring Boot plugin instead of through NetBeans. You may also choose to have NetBeans remove the Maven wrapper from the generated project.

Once you've set the final bit of project information, click Finish to generate the project and have it added to your NetBeans workspace.

A.4 Initializing a project at start.spring.io

Although one of the IDE-based initialization options described thus far will likely suit your needs, it's possible that you may use a completely different IDE, or you might favor working with a simpler text editor. In that case, you can still take advantage of the Spring Initializr using the Initializr web-based interface.

To get started, direct your favorite web browser to <https://start.spring.io>. You should see the simple version of the Spring Initializr web user interface, shown in figure A.15.

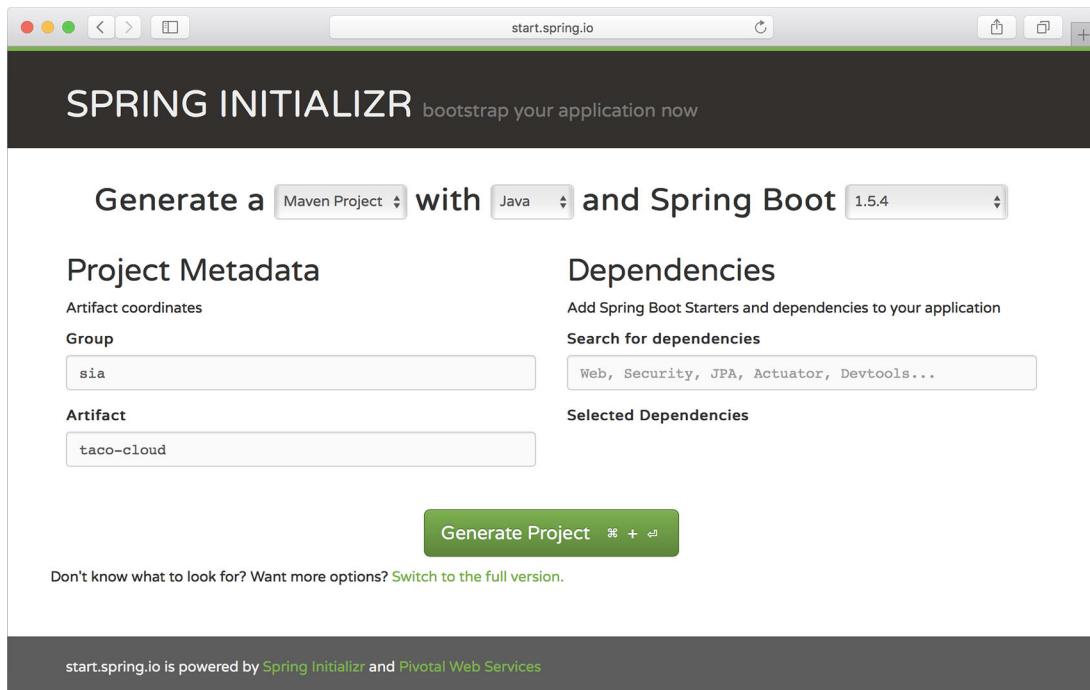


Figure A.15 The simple version of the Spring Initializr web interface

In the simple version of the Initializr web application, you're only asked for some very basic information, including whether you want to build with Maven or Gradle, which language you want to develop the project with, which version of Spring Boot to build against, and the group and artifact IDs of the project.

You'll also have the option of specifying dependencies by typing search criteria in the Search for Dependencies box. For example, as shown in figure A.16, you can type "web" to search for any dependencies where "web" is a keyword.

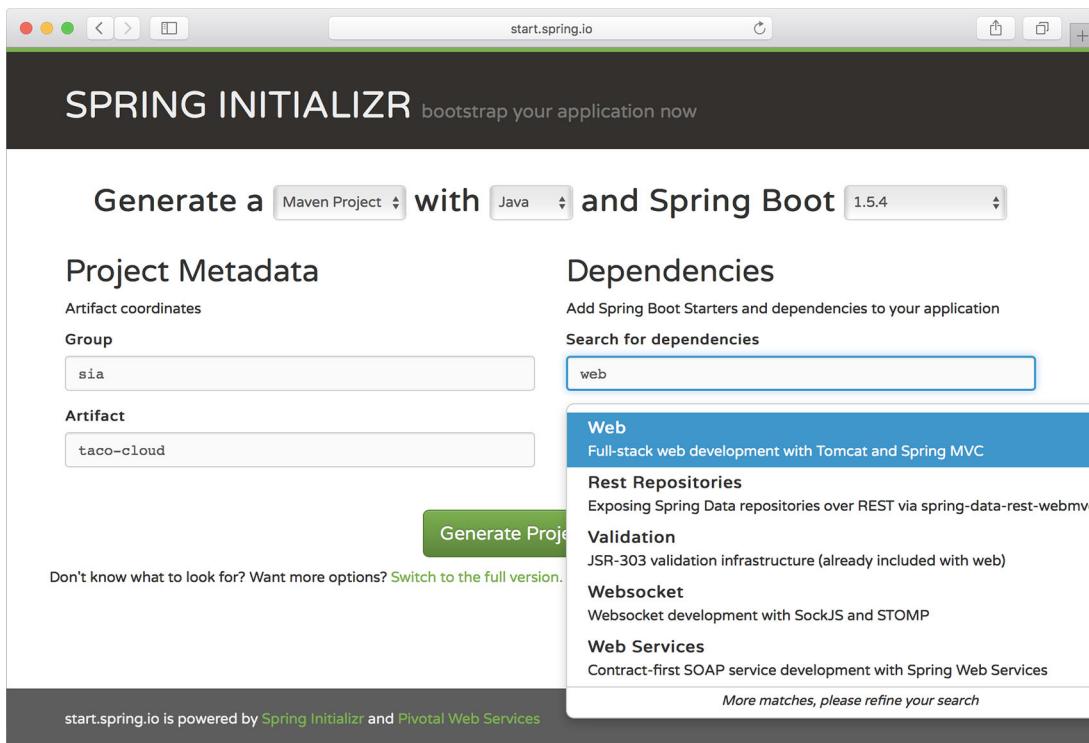


Figure A.16 Searching for dependencies

When you see the dependency you want, press Return on your keyboard to select it, and it will be added to the list of selected dependencies. The boxes beneath Selected Dependencies in figure A.17 show that the Web, Thymeleaf, DevTools, and Lombok dependencies have been selected.

If you decide you don't need a selected dependency, you can click the X to the right of the dependency entry to remove it.

When you're finished, you can click the Generate Project button (or use the keyboard shortcut displayed on the button, which will vary by operating system) to have the Initializr generate the project and download it as a zip file. Then you can unzip the project and load it in whatever IDE or editor you choose.

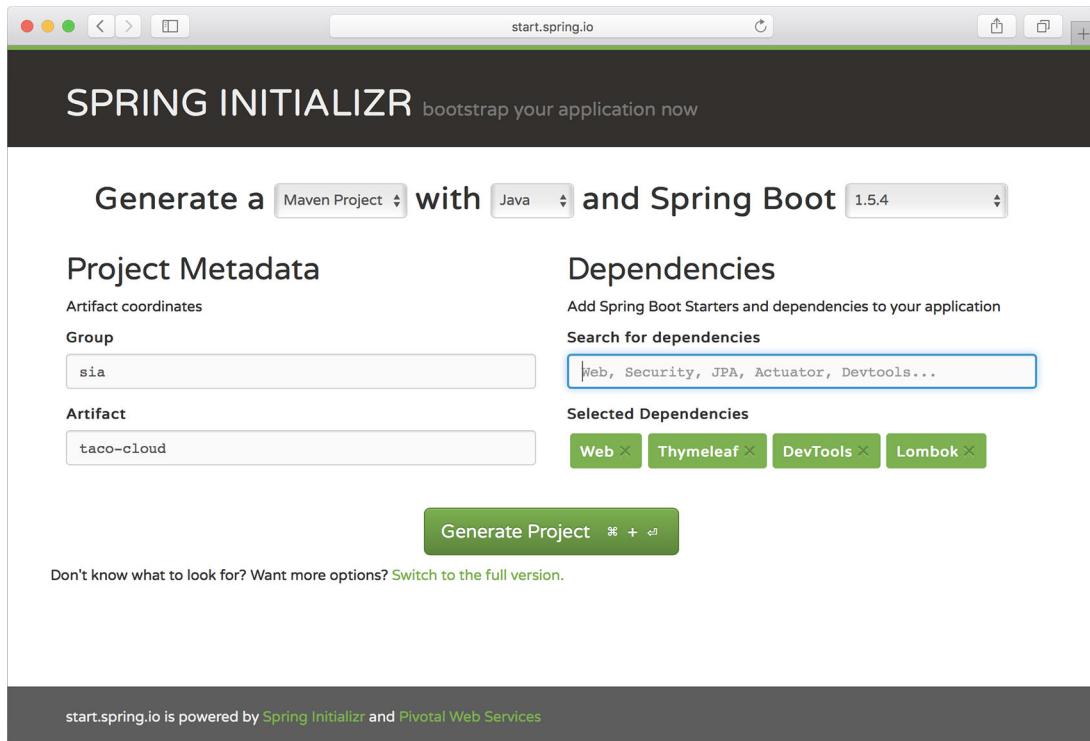


Figure A.17 Selecting dependencies

If you prefer a little more control, you can click the Switch to the Full Version link under Generate Project to expand the user interface with more fields and a complete listing of check boxes for all available dependencies. Figure A.18 shows a little bit of what the full version of the web interface looks like.

Most of the fields in the full version are derived from the Group and Artifact fields or have default values when you're using the simple version. The full version gives you the opportunity to override those derived/default values if you wish.

Figure A.18 only shows a small sample of the set of dependency check boxes that are available in the full version, so you might scroll a lot to find what you're looking for. Fortunately, the search box still works in the full version of the user interface.

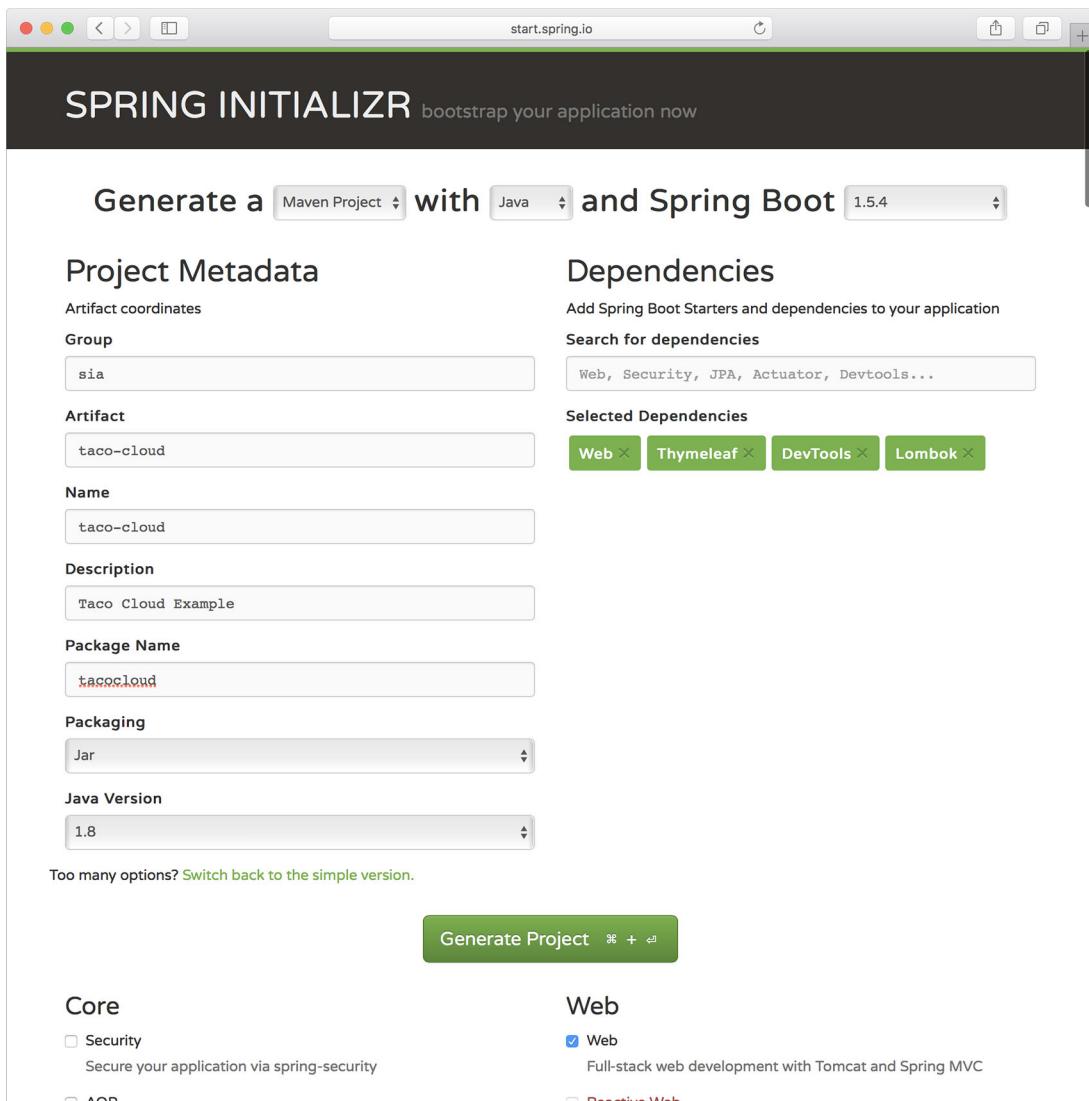


Figure A.18 The full version of the Initializr user interface

A.5 Initializing a project from the command line

The IDE and browser-based user interfaces for the Spring Initializr are probably the most common way that you'll bootstrap your projects. They're all just clients of a REST service offered by the Initializr application. In some special cases (for example, in a scripted scenario), you might find it useful to consume the Initializr service directly from the command line.

There are two ways to consume the API:

- Using the curl command (or some similar command-line REST client command)
- Using the Spring Boot Command Line Interface (aka, Spring Boot CLI)

Let's look at these options, starting with the curl command.

A.5.1 curl and the Initializr API

The simplest way to bootstrap a Spring project with curl is to consume the API like this:

```
% curl https://start.spring.io/starter.zip -o demo.zip
```

In this case, you're requesting the /starter.zip endpoint from the Initializr, which will generate a Spring project and download it as a zip file. The generated project will be Maven-built and will have no dependencies other than the base Spring Boot starter dependency. All project information in the project's pom.xml file will be set to default values.

If you don't specify otherwise, the name of the file will be starter.zip. But in this case, the -o option specifies that the downloaded file should be named demo.zip.

The publicly available Spring Initializr server is hosted at <https://start.spring.io>, but if you're using a custom Initializr, you'll need to adapt the given URL accordingly.

You'll probably want to specify a few more details and dependencies beyond the given defaults. Table A.1 lists all of the parameters (and their defaults) when consuming the Spring Initializr REST service.

Table A.1 Request parameters supported by the Initializr API

Parameter	Description	Default value
groupId	The project's group ID, for the sake of organization in a Maven repository.	com.example
artifactId	The project's artifact ID, as it would appear in a Maven repository.	demo
version	The project version.	0.0.1-SNAPSHOT
name	The project name. Also used to determine the name of the application's main class (with an Application suffix).	demo
description	The project description.	Demo project for Spring Boot
packageName	The project's base package name.	com.example.demo
dependencies	Dependencies to include in the project's build specification.	The base Spring Boot starter
type	The kind of project to generate. Either maven-project or gradle-project.	maven-project

Table A.1 Request parameters supported by the Initializr API (*continued*)

Parameter	Description	Default value
javaVersion	The version of Java to build with.	1.8
bootVersion	The version of Boot to build against.	The current GA version of Spring Boot
language	The programming language to use. Either java, groovy, or kotlin.	java
packaging	How the project should be packaged. Either jar or war.	jar
applicationName	The name of the application.	The value of the name parameter
baseDir	The name of the base directory in the generated archive.	The root directory

You can also get this list of parameters, as well as a list of available dependencies, by making a simple request to the base Initializr URL:

```
% curl https://start.spring.io
```

The dependencies parameter is the one you'll probably find the most useful. For example, suppose that you want to create a simple web project with Spring. The following command-line use of curl will produce a project zip with the web starter as a dependency:

```
% curl https://start.spring.io/starter.zip \
-d dependencies=web \
-o demo.zip
```

As a more complex example, suppose you wanted to develop a web application that uses Spring Data JPA for data persistence. You also want to build it with Gradle and the project should be under a directory named my-dir within the zip file. And let's suppose that rather than just download a zip file, you want the project unpacked into your filesystem upon download. In that case, the following command should do the trick:

```
% curl https://start.spring.io/starter.tgz \
-d dependencies=web,data-jpa \
-d type=gradle-project \
-d baseDir=my-dir | tar -xzvf -
```

Here, the downloaded zip file is piped to the tar command for unpacking.

A.5.2 Spring Boot command-line interface

The Spring Boot CLI is another option for initializing Spring applications. You can install the Spring Boot CLI in many ways, but probably the easiest way (and my favorite) is to use SDKMAN (<http://sdkman.io/>):

```
% sdk install springboot
```

Once the Spring Boot CLI is installed, you can start using it to generate projects, much like with curl. The command you'll use is `spring init`. In fact, the simplest way to use the Spring Boot CLI to generate a project is like this:

```
% spring init
```

This will result in a barebones Spring Boot project being downloaded in a zip file named `demo.zip`.

However, you'll probably want to specify more details and dependencies. Table A.2 lists all of the parameters available to the `spring init` command.

Table A.2 Request parameters supported by the `spring init` command

Parameter	Description	Default value
group-id	The project's group ID, for the sake of organization in a Maven repository.	com.example
artifact-id	The project's artifact ID, as it would appear in a Maven repository.	demo
version	The project version.	0.0.1-SNAPSHOT
name	The project name. Also used to determine the name of the application's main class (with an Application suffix).	demo
description	The project description.	Demo project for Spring Boot
package-name	The project's base package name.	com.example.demo
dependencies	Dependencies to include in the project's build specification.	The base Spring Boot starter
type	The kind of project to generate. Either <code>maven-project</code> or <code>gradle-project</code> .	<code>maven-project</code>
java-version	The version of Java to build with.	1.8
boot-version	The version of Boot to build against.	The current GA version of Spring Boot
language	The programming language to use. Either <code>java</code> , <code>groovy</code> , or <code>kotlin</code> .	java
packaging	How the project should be packaged. Either <code>jar</code> or <code>war</code> .	<code>jar</code>

You can also get this list of parameters, as well as a list of available dependencies, by using the `--list` parameter:

```
% spring init --list
```

Suppose you wish to create a web application that builds against Java 1.7. The following command uses the `--dependencies` and `--java-version` parameters to make those choices:

```
% spring init --dependencies=web --java-version=1.7
```

Or suppose you want to create a web application with Spring Data JPA for persistence, and you'd like to use Gradle to perform the build instead of Maven. You'd use the following command:

```
% spring init --dependencies=web,jpa --type=gradle-project
```

You may also notice that many of the `spring init` parameters are the same as or similar to the parameters for the `curl` option. That said, the `spring init` command doesn't support all of the same parameters as the `curl` option (`baseDir`, for example) and the parameters are hyphen-delimited instead of camel-cased (for example, `package-name` versus `packageName`).

A.6 *Creating Spring applications with a meta-framework*

It's also worth noting that there are a couple of frameworks that are built on top of Spring and Spring Boot:

- Grails—<https://grails.org/>
- JHipster—<https://jhipster.github.io/>

These meta-frameworks offer higher-level rapid development of Spring applications, while still offering everything that Spring and Spring Boot offer.

These meta-frameworks each offer their own unique development model and are, in fact, frameworks in their own right, so it wouldn't do them justice to simply present them as project initialization mechanisms in this appendix. Indeed, they each could have entire books written about them.

I won't delve into how to use these meta-frameworks to initialize a Spring project. Nevertheless, I include them here to make you aware that they are other ways to initialize and develop Spring applications.

A.7 *Building and running projects*

No matter how you initialize your project, you can always run the application from the command line with the `java -jar` command:

```
% java -jar demo.jar
```

This will even work if you decide to create a WAR file distribution instead of a JAR file:

```
% java -jar demo.war
```

You can also take advantage of the Spring Boot Maven and Gradle plugins to run your application. For example, if your project is built with Maven, you can run it like this:

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

If, on the other hand, you've chosen to build your project with Gradle, you can run your project like this:

```
% gradle bootRun
```

In either case, whether using Maven or Gradle, the build tool will first build your project (if it hasn't already been built) and run it.

index

Symbols

@ operator 38
@{} operator 38

A

AbstractMessageRouter 222
AbstractRepositoryEventListener 450–451
access() method 105
ActiveMQQueue 184
Actuator. *See* Spring Boot Actuator
addNote() method 425
addScript() method 115
addScripts() method 115
addViewControllers() method 52, 107
Advanced Message Queueing Protocol (AMQP) 179
all() method 266
@AllowFiltering annotation 312
AMQP (Advanced Message Queueing Protocol) 179
AmqpHeaderConverter 197
and() method 107
antMatchers() method 294
any() method 266
ApiProperties 236
APIs. *See* reactive APIs
applicationConfig property 406
applicationName parameter 482
applications. *See* Spring applications
artifactId parameter 481
asLink() method 176
asynchronous messaging 178–208
 with JMS 179–192
 JmsTemplate 181–188

receiving messages 188–192
setting up JMS 179–181
with Kafka 202–208
 sending messages with KafkaTemplate 204–206
 setting up 203–204
 writing Kafka listeners 206–208
with RabbitMQ 192–202
 adding to Spring 193–194
 receiving message from 198–202
 sending messages with RabbitTemplate 194–198
AtomicInteger 229
authentication. *See* user authentication,
 customizing
@AuthenticationPrincipal annotation 111
autoconfiguration, fine-tuning 115–121
 configuring data source 117–118
 configuring embedded server 119
 configuring logging 120–121
 environment abstraction 116–117
 using special property values 121
@Autowired annotation 61
autowiring 6
availableTags 415

B

base path, Actuator 397
baseDir parameter 482
@Bean annotation 5
bean wiring 115
bean wiring report 403–404
beans, conditionally creating with profiles 132–133
bill of materials (BOM) 248

- block() method 299
 BOM (bill of materials) 248
 bootstrapping applications 466–485
 building and running projects 484–485
 creating apps with meta-framework 484
 example 15–16
 initializing projects
 at start.spring.io 477–479
 from command line 480–484
 with IntelliJ IDEA 470–472
 with NetBeans 473–476
 with Spring Tool Suite 466–470
 bootVersion parameter 482
 browser refresh 24
 buffering data 263–266
 build specification 12–15
 BuildInfoContributor 418
-
- C**
- caching templates 54–55
 Cassandra 300–312
 data modeling 303–304
 enabling 301–303
 mapping domain types for persistence 304–309
 writing 309–312
 CassandraRepository interface 310
 ccExpiration property 47
 ccNumber property 47
 cf bind-service command 460
 channel adapters 228–230
 channel() method 216
 circuit breakers 376–383
 managing thresholds 382–383
 mitigating latency 381–382
 circuitBreaker.errorThresholdPercentage
 property 382
 circuitBreaker.requestVolumeThreshold
 property 382
 circuitBreaker.sleepWindowInMilliseconds
 property 383
 CLI (command-line interface) 481
 clients, Admin, registering 431–435
 discovering Admin client applications 433–435
 explicitly configuring Admin client
 applications 432–433
 Cloud Foundry 132, 454–461
 cloud. *See* Spring Cloud
 codecentric AG 430
 collections, creating reactive types from 250–251
 collectList() method 265
 Command Line Interface (CLI) 481
 command line, initializing projects from 480–484
 curl and Initializr API 481–482
 interface 483–484
- CommandLineRunner 133
 commandProperties attribute 381
 component scanning 6
 @ComponentScan annotation 15
 @ConditionalOnClass annotation 405
 @ConditionalOnMissingBean annotation 405
 Config Server 346
 config.client.version property 367
 configuration 343–375
 consuming shared configuration 352–353
 keeping configuration properties secret
 357–364
 encrypting properties in Git 357–360
 in Vault 360–364
 refreshing configuration properties on the
 fly 364–375
 automatically 367–375
 manually 365–367
 running configuration server 345–352
 enabling Config Server 346–349
 populating configuration repository 349–352
 serving application-specific properties 354–355
 serving properties from profiles 355–357
 sharing 344–345
 Spring Security 86–103
 customizing user authentication 96–103
 in-memory user store 88–89
 JDBC-based user store 89–92
 LDAP-backed user store 92–96
 @Configuration annotation 5
 configuration properties 114–133
 autoconfiguration, fine-tuning 115–121
 configuring data source 117–118
 configuring embedded server 119
 configuring logging 120–121
 environment abstraction 116–117
 using special property values 121
 defining holders 124–126
 metadata, declaring 126–129
 profiles 129–133
 activating 131–132
 conditionally creating beans with 132–133
 defining profile-specific properties 130–131
 configuration() method 292
 @ConfigurationProperties annotation 122–123,
 200, 232
 configure() method 99, 456
 configuredLevel property 410
 ContentTypeDelegatingMessageConverter 197
 contextSource() method 94
 contribute() method 417
 @Controller annotation 18
 controller class, creating 32–35
 controller, testing 20–21
 ControllerLinkBuilder 153

convertAndSend() method 182, 185, 195
copyToString() method 282
create keyspace command 302
createdAt property 59
createdDate property 165
createMessage() method 183
@CrossOrigin annotation 143
CrudRepository interface 81, 310, 318
CSRF (cross-site request forgery) 109–110
curl 7, 481–482
custom endpoints 165–167
custom hyperlinks 167–168
custom login page 106–108
customizing user authentication 96–103
 creating user-details service 98–100
 defining user domain and persistence 96–98
registering users 100–103

D

data 56–83
 persisting with Spring Data JPA 75–83
 adding Spring Data JPA to project 76
 annotating domain as entities 76–80
 customizing JPA repositories 81–83
 declaring JPA repositories 80–81
 reading and writing with JDBC 57–75
 adapting domain for persistence 59
 defining schema and preloading data 64–66
 inserting data 66–75
 working with JdbcTemplate 60–64
 source of, configuring 117–118
@Data annotation 32
data-backed services, enabling 160–168
 adding custom endpoints 165–167
 adding custom hyperlinks to Spring Data endpoints 167–168
 adjusting resource paths and relation names 162–164
 paging and sorting 164–165
databases, embedded 313
default user queries, overriding 90–91
defaultRequestChannel attribute 211
defaultSuccessUrl parameter 108
delayElements() method 254, 258
delaySubscription() method 254
delete() method 174
deleteById() method 149
@DeleteMapping annotation 148
deleteNote() method 425
deleteOrder() method 149
dependencies 476
dependency injection (DI) 4

deployment 454–465
 building and deploying WAR files 456–458
 options, deciding between 455–456
 pushing JAR files to Cloud Foundry 458–461
 running Spring Boot in Docker container 461–464
description parameter 481
DI (dependency injection) 4
diagramming reactive flows 246
@Digits annotation 48
DirectChannel 218
directory attribute 213
disable() method 110
disabling endpoints, Actuator 398
diskSpace 402
DispatcherServlet 456
displaying information 30–40
 creating controller class 32–35
 designing view 35–40
 establishing domain 31–32
Docker container, running Spring Boot in 461–464
@Document annotation 314
domain, establishing 31–32
doTransform() method 235
DSL, configuring integration flows using 215–216

E

email integration flows 231–237
EmailProperties class 232
embedded databases 313
embedded LDAP server, configuring 94–96
embedded relationships, naming 159–160
embedded server, configuring 119
EmptyResultDataAccessException 149
@EnableAdminServer annotation 431
@EnableAutoConfiguration annotation 15
@EnableConfigServer annotation 347
@EnableEurekaServer annotation 328
@EnableFeignClients annotation 340
@EnableHystrix annotation 379
@EnableHystrixDashboard annotation 384
@EnableWebFluxSecurity annotation 293
@EnableWebSecurity annotation 292
encoded passwords 91–92
encrypt.key property 357
endpoint modules 230–231
endpoints, Actuator
 consuming 399–416
 fetching essential application information 400–411
 tapping runtime metrics 413–416
 viewing application activity 411–413
enabling and disabling 398

endpoints, Spring Data
 adding custom hyperlinks to 167–168
 custom, adding 165–167
`@Entity` annotation 304
 entry points 462
 environment abstraction 116–117
 environment properties, Admin server 438
`equals()` method 31
 errors attribute 50
 Eureka service registry 326–334
 client-side load balancer 327–330
 configuring 330–332
 overview 326–327
 scaling 333–334
`eureka.client.fetch-registry` property 331
`eureka.client.register-with-eureka` property 331, 434
`eureka.client.service-url` property 331, 336
`eureka.instance.hostname` property 331
`eureka.server.enable-self-preservation`
 property 331–332
`evenChannel` 221
 event looping 270
`exchange()` method 171, 291
`execute()` method 73, 171
`executeAndReturnKey()` method 73
`execution.timeout.enabled` property 382
`ExecutorChannel` 218
`expectStatus()` method 281
`ExpressionInterceptUrlRegistry` 104

F

failure and latency 376–391
 aggregating multiple `Hystrix` streams 389–391
 circuit breakers 376–383
 managing circuit breaker thresholds 382–383
 mitigating latency 381–382
 monitoring failures 383–389
 `Hystrix` dashboard 384–387
 `Hystrix` thread pools 387–389
`Feign` library 340–342
`@FeignClient` annotation 341
`@Field` annotation 314
 FIFO (first in, first out) 218
`fileWriterChannel` 212
`FileWriterGateway` 211, 227
`FileWritingMessageHandler` 214, 230
`filter()` method 220, 259
 filtering data from reactive types 257–260
 filters 219–220
 final property 32
`findAll()` method 61–62, 143
`findByOrderByCreatedAtDesc()` method 319

`findByUser()` method 122
`findByUsername()` method 98, 295, 311, 320
`findByUserOrderByPlacedAtDesc()` method 123
`findOne()` method 61–62
 first in, first out (FIFO) 218
`flatMap()` method 261, 291
 Flux data 251–252
`FluxMessageChannel` 218
`follow()` method 176
`formLogin()` method 106
 forms
 processing submission 40–45
 validating input 45–51
 declaring validation rules 46–48
 displaying validation errors 49–51
 performing validation at form binding 48–49
 frameworkless framework 25
`FreeMarker` 53
`from()` method 229
`fromArray()` method 250, 299
`fromIterable()` method 251, 299
`fromStream()` method 299

G

gateways 227–228
`@GeneratedValue` annotation 78
`GenericHandler` 226
`GenericTransformer` 214
 GET requests
 handling 34–35
 testing 279–282
`getAuthorities()` method 97
`getContent()` method 273
`getForEntity()` method 172–173
`getForObject()` method 172–173, 285, 338
`getHref()` method 177
`getImapUrl()` method 234
`@GetMapping` annotation 18, 35, 139
`getMessageConverter()` method 195
`getPrincipal()` method 112
 Git
 authenticating with backend 352
 encrypting properties in 357–360
 serving configuration from subpaths 350–351
`Gogs` notification extractor 373–374
`Grails` 484
`greeting.message` property 367
`Groovy Templates` 53
`groupId` parameter 481
`groupSearchBase()` method 93
`groupSearchFilter()` method 92

H

H2 console 25
handle() method 207, 236
hasErrors() method 49
hashCode() method 31
hasRole() method 104
HATEOAS (Hypermedia as the Engine of Application State) 149, 399
helloRouterFunction() method 278
home() method 18
href property 175
HTTP 404 (Not Found) error 35, 290
HTTP requests
 mappings 408–409
 tracing 441–442
HTTPie 144
Hypermedia as the Engine of Application State (HATEOAS) 149, 399
hypermedia, enabling 149–160
 adding hyperlinks 152–154
 creating resource assemblers 154–158
 naming embedded relationships 159–160
Hystrix
 dashboard 384–387
 streams, aggregating multiple 389–391
 thread pools 387–389
@HystrixCommand annotation 378

I

IAAS (Infrastructure-as-a-Service) 461
@Id annotation 77, 314
imperative code 242
@ImportResource annotation 213
inboundAdapter() method 229
@InboundChannelAdapter annotation 229
incognito mode 88
/info endpoint, contributing information to 416–420
InfoContributor interface 416
Infrastructure-as-a-Service (IAAS) 461
initializing applications 6–17
initializing projects
 at start.spring.io 477–479
 from command line 480–484
 curl and Initializr API 481–482
 interface 483–484
 with IntelliJ IDEA 470–472
 with NetBeans 473–476
 with Spring Tool Suite 7–11, 466–470
Initializr API 481–482
in-memory user store 88–89
inMemoryAuthentication() method 89
instantiateResource() method 156

integration. *See* Spring Integration
IntegrationFlows class 216
IntelliJ IDEA, initializing projects with 470–472
interval() method 252
InventoryService bean 5

J

Jackson2JsonMessageConverter 197
JAR files, pushing to Cloud Foundry 458–461
Java Database Connectivity. *See* JDBC
Java Management Extensions. *See* JMX
Java Message Service. *See* JMS
Java Persistence API. *See* Spring Data JPA
Java Streams 244
Java virtual machine (JVM) 24
Java, configuring integration flows in 213–215
JavaServer Pages (JSP) 35
javaVersion parameter 482
JDBC (Java Database Connectivity) based user store 89–92
 overriding default user queries 90–91
 working with encoded passwords 91–92
JDBC (Java Database Connectivity), reading and writing data with 57–75
 adapting domain for persistence 59
 defining schema and preloading data 64–66
 inserting data 66–75
 working with JdbcTemplate 60–64
jdbcTemplateAuthentication() method 90
JdbcTemplate class 26, 57
JHipster 484
JMS (Java Message Service) 179–192
 JmsTemplate 181–188
 configuring message converter 185–187
 converting messages before sending 185
 post-processing messages 187–188
 receiving messages 188–192
 declaring message listeners 191–192
 with JmsTemplate 189–191
 setting up 179–181
 @JmsListener annotation 191, 202
 JmsOrderMessagingService 183
 jms.send() method 183
 JmsTemplate 181–188
 configuring message converter 185–187
 converting messages before sending 185
 post-processing messages 187–188
 receiving messages with 189–191
JMX (Java Management Extensions) 446–453
 Actuator MBeans 446–449
 creating MBeans 449–451
 sending notifications 451–453
JPA (Java Persistence API). *See* Spring Data JPA (Java Persistence API)

jsonPath() method 281
 JSP (JavaServer Pages) 35
 just() method 246
 JVM (Java virtual machine) 24

K

Kafka 202–208
 sending messages with KafkaTemplate 204–206
 setting up 203–204
 writing listeners 206–208

L

latency. *See* failure and latency
 LDAP (Lightweight Directory Access Protocol)
 backed user store 92–96
 configuring embedded LDAP server 94–96
 configuring password comparison 93–94
 referring to remote LDAP server 94
 LDIF (LDAP Data Interchange Format) 95
 links property 151
 linkTo() method 154
 LiveReload 24
 @LoadBalanced annotation 338
 loadByUsername() method 99
 log() method 264
 Logback 120
 loggerLevels 448
 logging
 configuring 120–121
 out 109
 logging.file property 121
 logging.path property 121
 loginPage() method 107
 Lombok 32, 478

M

Mail imapInboundAdapter() method 234
 main() method 16
 @ManagedAttribute annotation 449
 @ManagedOperation annotation 449
 @ManagedResource annotation 449
 management.endpoint.health.show-details
 property 401
 management.endpoints.web.exposure.exclude
 property 398
 management.endpoints.web.exposure.include
 property 398
 management.endpoint.web.base-path
 property 397
 @ManyToMany annotation 78
 map() method 246
 mapping reactive data 260–263

MappingJackson2MessageConverter 186
 mapRowToIngredient() method 58
 MarshallingMessageConverter 197
 matches() method 92
 @Max annotation 126
 MBeans
 Actuator MBeans 446–449
 creating 449–451
 MDBs (message-driven beans) 179
 mergeWith() method 253
 message channels 217–219
 message-driven beans (MDBs) 179
 MessagePostProcessor 182, 188, 198
 MessageProperties 196
 messaging. *See* asynchronous messaging
 @MessagingGateway annotation 211
 MessagingMessageConverter 197
 metadata, declaring 126–129
 MeterRegistry 422
 methodOn() method 154
 Micrometer 422
 microservices 323–342
 Eureka service registry 326–334
 client-side load balancer 327–330
 configuring 330–332
 configuring Eureka client properties 335–337
 overview 326–327
 scaling 333–334
 overview 324–326
 registering and discovering services 334–342
 configuring Eureka client properties
 335–337
 consuming services 337–342
 @Min annotation 126
 @ModelAttribute annotation 70
 MongoDB. *See* Spring Data MongoDB
 MongoRepository 318
 MongoTemplate bean 405
 monitoring Spring. *See* JMX
 monitoring threads, Admin server 440
 MPA (multipage application) 139
 Mustache 53

N

name parameter 481
 NetBeans, initializing projects with 473–476
 NetworkTopologyStrategy 302
 ngOnInit() method 140
 @NoArgsConstructor annotation 77
 Not Found (HTTP 404) error 35, 290
 @NotBlank annotation 47
 notes() method 425
 NotesEndpoint class 424
 NotificationPublisherAware interface 451

@NotNull annotation 46
numberChannel 220

O

objects, creating reactive types from 249–250
oddChannel 221
onAfterCreate() method 423, 451
onError() method 245
onNext() method 245
onStatus() method 289
onSubmit() method 145
onSubscribe() method 244
OpenFeign 340
OrderController class 43
orderForm() method 42
orderInserter variable 72
ordersForUser() method 122
OrderSplitter 224
overriding default user queries 90–91

P

PaaS (platform-as-a-service) 465
packageName parameter 481
pageSize property 124
paging 164–165
PagingAndSortingRepository 311
ParameterizedTypeReference 176, 201
passwordCompare() method 93
passwordEncoder() method 91, 94
passwords
 comparison of, configuring 93–94
 encoded 91–92
patchOrder() method 148
pathMatchers() method 294
@PathVariable annotation 144, 341
@Pattern annotation 48
PayloadTypeRouter 224
permitAll() method 104
Pivotal Web Services (PWS) 459
placedAt property 73, 122
platform-as-a-service (PaaS) 465
poChannel 224
POST requests, testing 282–283
postForEntity() method 174
postForLocation() method 174–175
postForObject() method 174–175
@PostMapping annotation 35, 139
PreparedStatementCreator() method 69
@PrePersist annotation 78
PriorityChannel 218
processDesign() method 40, 46, 70
processOrder() method 46
processRegistration() method 102

ProductService bean 5
profiles 129–133
 activating 131–132
 conditionally creating beans with 132–133
 defining profile-specific properties 130–131
projects. *See* Spring projects
property injection 115
propertySources property 349
PublishSubscribeChannel 217
pull model 188
push model 188
put() method 173
PWS (Pivotal Web Services) 459

Q

query() method 63
queryForObject() method 58, 62–63
QueueChannel 218

R

RabbitMQ 192–202
 adding to Spring 193–194
 receiving message from 198–202
 handling messages with listeners 201–202
 with RabbitTemplate 198–201
 sending messages with RabbitTemplate 194–198
 configuring message converter 197
 setting message properties 197–198
RabbitTemplate beans 193
range() method 251
reactive APIs 269–295
 consuming REST APIs reactively 285–292
 deleting resources 288–289
 exchanging requests 290–292
 GETting resources 285–287
 handling errors 289–290
 sending resources 287–288
 functional request handlers, defining 276–279
 reactive controllers
 testing 279–285
 writing 272–276
 securing reactive web APIs 292–295
 configuring reactive user details service 294–295
 configuring reactive web security 292–294
 Spring WebFlux 269–276
reactive code 242
reactive programming
 defining Reactive Streams 243–245
 overview 242–245
 See also Reactor
reactive repositories. *See* Spring Data

ReactiveCassandraRepository interface 309
 ReactiveCrudRepository interface 309, 318
 ReactiveMongoRepository 318
 ReactiveUserDetailsService 295
 Reactor 241–268
 getting started with 245–248
 adding dependencies 247–248
 diagramming reactive flows 246
 reactive streams, transforming and
 filtering 257–266
 buffering data 263–266
 filtering data from reactive types 257–260
 mapping reactive data 260–263
 reactive types
 combining 253–256
 performing logic operations on 266–268
 reading and writing data, with JDBC 57–75
 adapting domain for persistence 59
 defining schema and preloading data 64–66
 inserting data 66–75
 inserting data with SimpleJdbcTemplate 71–75
 saving data with JdbcTemplate 67–71
 working with JdbcTemplate 60–64
 defining JDBC repositories 60–62
 inserting row 62–64
 receive() method 190, 199
 receiveAndConvert() method 190, 199
 recent links 151
 registerForm() method 101
 registering
 clients, Admin 431–435
 discovering Admin client applications 433–435
 explicitly configuring Admin client
 applications 432–433
 users 100–103
 RegistrationController class 100
 relation names, adjusting 162–164
 release train 297
 remote LDAP server, referring to 94
 RendezvousChannel 218
 replacement operation 146
 @Repository annotation 61
 @RepositoryRestController annotation 166–167
 @RequestBody annotation 146
 @RequestMapping annotation 34–35, 143
 RequestPredicate 278
 @RequiredArgsConstructor annotation 77
 resource assemblers, creating 154–158
 resource paths, adjusting 162–164
 ResourceProcessor 168
 ResourceSupport 155
 @ResponseStatus annotation 149
 ResponseStatusException 416

REST services 137–168
 consuming 169–177
 navigating REST APIs with Traverson 175–177
 REST APIs, reactively 285–292
 REST endpoints with RestTemplate 175–177
 data-backed services, enabling 160–168
 adding custom endpoints 165–167
 adding custom hyperlinks to Spring Data
 endpoints 167–168
 adjusting resource paths and relation
 names 162–164
 paging and sorting 164–165
 hypermedia, enabling 149–160
 adding hyperlinks 152–154
 creating resource assemblers 154–158
 naming embedded relationships 159–160
 RESTful controllers, writing 138–149
 deleting data from server 148–149
 retrieving data from server 140–144
 sending data to server 145–146
 updating data on server 146–148
 restart, automatic 23–24
 RestClientException 380
 @RestController annotation 142, 160
 @RestResource annotation 164
 RestTemplate
 consuming REST endpoints with 175–177
 delete() method 174
 getForObject() and getForEntity()
 methods 172–173
 postForObject() and postForLocation()
 methods 174–175
 put() method 173
 consuming services with 337–339
 retrieve() method 290
 romanNumberChannel 220
 routerFunction() method 278–279
 routers 221–223
 run() method 16
 @RunWith annotation 16, 284
 RXJava types 275

S

save() method 67
 saveAll() method 276, 298
 scanning 6, 15
 SDKMAN 483
 securing reactive web APIs 292–295
 configuring reactive user details service 294–295
 configuring reactive web security 292–294
 security. *See* Spring Security

SecurityConfig class 88
SecurityContextHolder 111
security.user.name property 117
security.user.password property 117, 127
securityWebFilterChain() method 293
self links 151
send() method 182, 195
sendDefault() method 205
sendNotification() method 451
SerializerMessageConverter 197
server, Admin 435–442
 creating 430–431
 examining environment properties 438
 monitoring threads 440
 securing 442–445
 authenticating with Actuator 444–445
 enabling login 443–444
 tracing HTTP requests 441–442
 viewing and setting logging levels 439
 viewing general application health and
 information 436–437
 watching key metrics 437–438
ServerHttpSecurity object 293
server.port property 119, 355, 385
server.ssl.key-store property 119
service activators 225–227
@ServiceActivator annotation 214, 226
@SessionAttributes annotation 70
SessionStatus parameter 75
setExpectReply() method 214
setHeader() method 198
setNotificationPublisher() method 451
setStringProperty() method 187
setIdMappings() method 186
setTypeIdPropertyName() method 186
setViewName() method 52
showDesignForm() method 34–35
SimpleJdbcInsert class 66
SimpleJdbcTemplate 71–75
SimpleMessageConverter 186, 197
single-page application (SPA) 139
@Size annotation 46
skip() method 257
slash() method 153
@Slf4j annotation 34
sorting 164–165
sources() method 457
SPA (single-page application) 139
special property values 121
SpEL (Spring Expression Language) 212
splitOrderChannel 224
splitters 223–225
Spring applications
 automatic restart 23–24
 initializing 6–17
testing 16–17
writing 17–26
 building and running application 21–23
 defining view 19
 handling web requests 17–19
 Spring Boot DevTools use 23–25
 testing controller 20–21
 See also bootstrapping applications
Spring Batch 27–28
Spring Boot
 overview of 26–27
 running in Docker container 461–464
Spring Boot Actuator 395–428
 base path, configuring 397
 customizing 416–426
 contributing information to /info
 endpoint 416–420
 creating custom endpoints 424–426
 defining custom health indicators 421–422
 registering custom metrics 422–424
 endpoints, consuming 399–416
 fetching essential application
 information 400–411
 tapping runtime metrics 413–416
 viewing application activity 411–413
 endpoints, enabling and disabling 398
 MBeans 446–449
 overview 396–397
 securing 426–428
Spring Boot Admin 429–445
 clients, registering 431–435
 overview 430
 server 435–442
 creating 430–431
 examining environment properties 438
 monitoring threads 440
 securing 442–445
 tracing HTTP requests 441–442
 viewing and setting logging levels 439
 viewing general application health and
 information 436–437
 watching key metrics 437–438
Spring Boot DevTools 23–25
 automatic application restart 23–24
 automatic browser refresh 24
 built in H2 console 25
 template cache disable 24
Spring Cloud 28
Spring Data 27, 296–320
 converting between reactive and nonreactive
 types 298–300
 developing reactive repositories 300
 reactive Cassandra repositories 300–312
 data modeling 303–304
 enabling Cassandra 301–303

- Spring Data (*continued*)**
- mapping domain types for Cassandra persistence 304–309
 - writing 309–312
 - writing reactive MongoDB repositories 312–320
 - enabling Spring Data MongoDB 312–314
 - mapping domain types to documents 314–317
 - writing reactive MongoDB repository interfaces 317–320
 - Spring Data JPA (Java Persistence API) 75–83
 - adding to project 76
 - annotating domain as entities 76–80
 - customizing JPA repositories 81–83
 - declaring JPA repositories 80–81
 - Spring Data MongoDB 312–320
 - enabling 312–314
 - mapping domain types to documents 314–317
 - writing reactive repository interfaces 317–320
 - Spring Expression Language (SpEL) 212
 - Spring Framework 26
 - spring init command 483
 - Spring Integration 27–28, 209–237
 - channel adapters 228–230
 - creating email integration flow 231–237
 - endpoint modules 230–231
 - filters 219–220
 - gateways 227–228
 - integration flows 210–216
 - configuring in Java 213–215
 - defining with XML 211–213
 - using DSL configuration 215–216
 - message channels 217–219
 - routers 221–223
 - service activators 225–227
 - splitters 223–225
 - transformers 220–221
 - Spring overview 4–6
 - Spring projects
 - initializing
 - at start.spring.io 477–479
 - from command line 480–484
 - with IntelliJ IDEA 470–472
 - with NetBeans 473–476
 - with Spring Tool Suite 7–11, 466–470
 - structure 11–17
 - bootstrapping application 15–16
 - build specification 12–15
 - SpringRunner 17
 - testing application 16–17
 - Spring Security 27, 84–113
 - configuring 86–103
 - customizing user authentication 96–103
 - in-memory user store 88–89
 - JDBC-based user store 89–92
 - LDAP-backed user store 92–96
 - enabling 85–86
 - knowing user 110–113
 - securing web requests 103–110
 - creating custom login page 106–108
 - CSRF forgery 109–110
 - logging out 109
 - securing requests 104–106
 - Spring Tool Suite
 - initializing projects with 466–470
 - initializing Spring projects with 7–11
 - Spring WebFlux 269–276
 - overview 271–272
 - reactive controllers, writing 272–276
 - handling input reactively 275–276
 - returning single values 274–275
 - RXJava types 275
 - writing reactive controllers 272–276
 - handling input reactively 275–276
 - returning single values 274–275
 - RXJava types 275
 - spring.activemq.broker-url property 181
 - spring.activemq.in-memory property 181
 - spring.active.profiles property 353
 - SpringApplication class 16
 - spring.application.name property 335, 353, 433
 - spring.boot.admin.client.metadata.user.name property 444
 - spring.boot.admin.client.metadata.user.password property 444
 - spring.boot.admin.client.url property 432
 - @SpringBootApplication annotation 15
 - @SpringBootConfiguration annotation 15
 - SpringBootServletInitializer 456
 - @SpringBootTest annotation 17, 284
 - spring.cassandra.contact-points property 302
 - spring.cassandra.keyspace-name property 302
 - spring.cassandra.port property 302
 - spring.cloud.config.discovery.enabled property 353
 - spring.cloud.config.server.encrypt.enabled property 359
 - spring.cloud.config.server.git.default-label property 351
 - spring.cloud.config.server.git.search-paths property 351
 - spring.cloud.config.server.git.uri property 348
 - spring.cloud.config.token property 364
 - spring.cloud.config.uri property 352
 - spring-cloud.version property 328
 - spring.data.mongodb.database property 314
 - spring.data.mongodb.host property 314
 - spring.data.mongodb.password property 314, 358

spring.data.mongodb.port property 314
 spring.data.mongodb.username property 314
 spring.datasource.data property 118
 spring.datasource.driver-class-name property 118
 spring.datasource.jndi-name property 118
 spring.datasource.schema property 118
 spring.jms.template.default-destination property 184
 SpringJUnit4ClassRunner 17
 spring.kafka.bootstrap-servers property 204
 spring.kafka.template.default-topic property 206
 spring.main.web-application-type property 237
 spring.profiles property 130
 spring.profiles.active property 131
 spring.rabbitmq.template.receive-timeout property 201
 SpringRunner 17
 SQLException 58
 StandardPasswordEncoder 91
 start.spring.io, initializing projects at 477–479
 StepVerifier 249
 StreamUtils 282
 subscribe() method 244
 System.currentTimeMillis() method 205
 systemEnvironment property 407

T

@Table annotation 304
 take() method 258
 template cache disable 24
 testHomePage() method 21
 testing
 applications 16–17
 controller 20–21
 reactive controllers 279–285
 testing GET requests 279–282
 testing POST requests 282–283
 testing with live server 284–285
 textInChannel 214
 threads, monitoring 440
 Thymeleaf 14, 18–19, 478
 timeout() method 287
 toAnyEndpoint() method 428
 toIterable() method 299
 Tomcat 22
 toObject() method 176
 toResource() method 156
 toRoman() method 220
 toString() method 31
 toUser() method 103
 tracing HTTP requests, Admin server 441–442
 @Transformer annotation 220
 transformerFlow() method 221
 transformers 220–221

Traverson library, navigating REST APIs with 175–177
 turbine.app-config property 390

U

UDT (user-defined type) 306
 UI (user interface) 429
 update() method 63, 66
 uppercase() method 228
 url() method 94
 user authentication, customizing 96–103
 creating user-details service 98–100
 defining user domain and persistence 96–98
 registering users 100–103
 user interface (UI) 429
 user queries, default 90–91
 user-defined type (UDT) 306
 @UserDefinedType annotation 307
 user-details service, creating 98–100
 userDetailsService() method 99, 295
 UsernameNotFoundException 98
 userPassword attribute 93
 UserRepositoryUserDetailsService 99
 userSearchBase() method 93
 userSearchFilter() method 92

V

@Valid annotation 48
 @Validated annotation 126
 validating form input 45–51
 declaring validation rules 46–48
 displaying validation errors 49–51
 performing validation at form binding 48–49
 Vault 360–364
 enabling backend in Config Server 362–363
 setting token in Config Server clients 364
 starting server 360–361
 writing application and profile-specific secrets 364
 writing secrets to 361–362
 version parameter 481
 view controllers 51–52
 view template library, choosing
 caching templates 54–55
 general discussion 52–55
 view, designing 35–40

W

WAR files, building and deploying 456–458
 web applications, developing 29–55
 displaying information 30–40
 creating controller class 32–35

web applications, developing (*continued*)
 designing view 35–40
 establishing domain 31–32
processing form submission 40–45
validating form input 45–51
 declaring validation rules 46–48
 displaying validation errors 49–51
 performing validation at form binding
 48–49
view controllers 51–52
 view template library, choosing 52–55
Web dependency 478
web requests, securing 103–110
 creating custom login page 106–108
 logging out 109
 preventing cross-site request forgery
 (CSRF) 109–110
WebApplicationInitialier 456
WebClient, consuming services with 339–340
WebClientResponseException 289
webEnvironment attribute 284
webhooks
 creating 368–371
 handling webhook updates in Config
 Server 371–372
 overview of 367
WebMvcConfigurer interface 51
@WebMvcTest annotation 21, 52

WebSecurityConfigurerAdapter class 88, 292, 427
WebTestClient 281
withDetail() method 417
withUser() method 89
wrapper scripts, Maven 11
writeToFile() method 211
writing data. *See* reading and writing data, with
 JDBC
writing Spring applications 17–26
 building and running application 21–23
 defining view 19
 handling web requests 17–19
Spring Boot DevTools use 23–25
 automatic application restart 23–24
 automatic browser refresh and template cache
 disable 24
 built in H2 console 25
testing controller 20–21

X

X-Config-Token header 363
XML, defining integration flows using 211–213
X-Vault-Token header 363

Z

zip() method 254

Spring IN ACTION Fifth Edition

Craig Walls

Spring Framework makes life easier for Java developers. New features in Spring 5 bring its productivity-focused approach to microservices, reactive development, and other modern application designs. With Spring Boot now fully integrated, you can start even complex projects with minimal configuration code. And the upgraded WebFlux framework supports reactive apps right out of the box!

Spring in Action, Fifth Edition guides you through Spring's core features, explained in Craig Walls' famously clear style. You'll roll up your sleeves and build a secure database-backed web app step by step. Along the way, you'll explore reactive programming, microservices, service discovery, RESTful APIs, deployment, and expert best practices. Whether you're just discovering Spring or leveling up to Spring 5, this Manning classic is your ticket!

What's Inside

- Building reactive applications
- Spring MVC for web apps and RESTful web services
- Securing applications with Spring Security
- Covers Spring 5.0

For intermediate Java developers.

Craig Walls is a principal software engineer at Pivotal, a popular author, an enthusiastic supporter of Spring Framework, and a frequent conference speaker.

To download their free eBook in PDF, ePUB, and Kindle formats,
owners of this book should visit
manning.com/books/spring-in-action-fifth-edition

Over 100,000 copies sold!



MANNING

\$49.99 / Can \$65.99 [INCLUDING eBook]

Free eBook

See first page

“This new edition is a comprehensive update that strikes the balance between practical instruction and comprehensive theory.”

—Daniel Vaughan
European Bioinformatics Institute

“The go-to book for learning the Spring Framework and an excellent reference guide.”

—Colin Joyce, Cisco

“Everything you need to know about Spring and how to build cloud-native applications.”

—David Witherspoon, Parsons

“This book is the Spring developer’s Swiss Army knife!”

—Riccardo Noviello
Nuvio Software Solutions

ISBN-13: 978-1-61729-494-5
ISBN-10: 1-61729-494-2



54999