

# HACKING WITH SPRING BOOT 2.4

CLASSIC EDITION

GREG L.  
TURNQUIST  
FOREWORD BY DR. DAVE SYER



**HACKING WITH  
SPRING BOOT 2.4**

CLASSIC EDITION

GREG L.  
TURNQUIST

FOREWORD BY DR. DAVE SYER

# TABLE OF CONTENTS

---

[Copyright](#)

[Credits](#)

[Dedication](#)

[Foreword](#)

[About the Author](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who should read this book](#)

[Optimal viewing](#)

[Conventions](#)

[Reader feedback](#)

[Support](#)

[Downloading the code](#)

[1](#)

[Building a Web App with Spring Boot](#)

What is Spring Boot?

Say hello to Spring MVC

Building an E-Commerce Platform with Spring Boot

Project Parent

Application Metadata

Spring Boot Starters

Spring Boot Maven Plugin

Initial Code

Autoconfiguration

Component Scanning

Creating a Spring MVC Controller

Tiptoeing into Templates

Summary

2

Data Access with Spring Boot

Defining Your E-Commerce App's Domain

Creating a Repository

Loading Test Data

Showing the cart

Adding items to a cart

Wrapping things inside a service

Querying the database

When query derivation isn't enough

Query by Example

Trade offs

Summary

3

Developer Tools for Spring Boot

Starting your application...faster

Say "hi" to Developer Tools

Automatic restarts and reloads

Exclusion of static resources

Disabling caches in debug mode

Logging extra web activity

Logging of changes in autoconfiguration

LiveReload support

Summary

4

Testing with Spring Boot

Writing unit tests

Running embedded container tests

Using Spring Boot's slice testing

## Summary

5

### Operations with Spring Boot

#### Deploying your application to production

Going to production with an über JAR

Going to production with Docker

Going to production with GraalVM

#### Managing your application in production

Pinging your app with Spring Boot Actuator and /actuator/health

Serving valuable app details with /actuator/info

Accessing additional actuator endpoints

Accessing loggers with /actuator/loggers

Reading operational data

Reading up on threads with /actuator/threaddump

Analyzing data using /actuator/heapdump

Tracing HTTP calls with /actuator/httptrace

Other operational readouts

Customizing management service routes

## Summary

6

## Building APIs with Spring Boot

Creating an HTTP web service

Creating an API portal

The challenges of API evolution

Creating a hypermedia-based web service

The value of hypermedia

Adding affordances to your API

Summary

7

## Messaging with Spring Boot

Picking your favorite solution

Tackling a problem with a familiar pattern

Testing with ease

Coding a solution

Crafting a test case

Coding a consumer

Summary

8

## Securing your Application with Spring Boot

Getting started

Getting real

[Taking the driver's seat](#)

[Tapping into user context](#)

[Method-level security](#)

[OAuth security](#)

[Summary](#)

[About the Author](#)

# COPYRIGHT

---

**Hacking with Spring Boot 2.4:**

*Classic Edition*

© 2021 Greg L. Turnquist

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

All opinions stated are the author's and do not necessarily reflect those of the Spring team, VMware, Dell EMC, or any other entity. Any usage of "Spring", "Spring Boot", "Spring

Framework", "Project Reactor", any other portfolio project, "VMware", "Testcontainers", "RabbitMQ", "Docker", "MongoDB", or any other entity is completely unaffiliated with its owners. Any excerpts and usages thereof are strictly in the spirit of "fair use."

Certain outputs have been edited to fit the format of this book. Nothing of critical value has been left out on purpose.

All rendering and typesetting performed by Asciidoctor, Asciidoctor PDF, and Asciidoctor EPUB3. Fonts include Noto Serif for prose and Fira Code for code. Special thanks to Dan Allen and the Asciidoctor community for their support in making this book possible.

Published: April 2021

Greg L. Turnquist  
c/o Hacking with Spring Boot  
P.O. Box 4042  
Clarksville, TN 37043  
USA

[GregLTurnquist.com/hacking-with-spring-boot-2.4-classic-edition](http://GregLTurnquist.com/hacking-with-spring-boot-2.4-classic-edition)

# CREDITS

---

*Author*

Greg L. Turnquist

*Foreword*

Dr. Dave Syer

*Cover*

AMDesign Studios

*Reviewers*

Sergei Egorov

Martin Fürstenau

Chua Wen Ching

# DEDICATION

---

*This book is dedicated to my YouTube fans.  
Your words and support have encouraged me to pour my heart  
and soul into video content that you can enjoy.*

# FOREWORD

---

Spring Boot has been such a success that it's probably not wrong now to describe it in 2021 as "mainstream". Practically every Java developer will know something about it, and many, maybe even the majority, will have used it in anger. But there's always something new to learn, and there are always new problems to solve in software engineering - that's what makes it so rewarding in the end. There's always something new to invent, too, and having the skill and opportunity to create code is extremely rewarding, intellectually and in other ways. One of the goals of Spring Boot is shared with the author of this book, and that is to get your ideas down into code as quickly and efficiently as possible, so you can get it to the most special place: Production. I wish you a short and pleasant journey, or maybe a long series of short and pleasant journeys.

In this book Greg has used his insider advantage to add Spring Boot knowledge to some old, well-seasoned favourite problems that you all will have experienced as Java developers. What better way to learn than to look at it through the lens of tasks that we all have to solve, nearly every day: creating HTTP

endpoints, securing them, connecting to databases, writing tests, sending messages? The book adds some new angles to these old chestnuts by applying some modern ideas and tools, so read it and you will learn about things like hypermedia and OpenID, all from the most practical and pragmatic of standpoints.

There are more things to Spring Boot than just main methods, embedded containers, autoconfiguration and management endpoints. The pure joy of getting started with a fully featured Spring application in a few lines of code cannot be understated, for instance. I invite you to dip into this book, break out an editor or an IDE and crank up some applications for yourself. Greg has been an important member of the Spring Boot team, despite having a day job doing other things in the Spring Engineering effort, and we can be grateful for that as well as the effort he has lavished on this excellent book. He has always been an educator and an informer, as well as an engineer, and this shows through very clearly in the book. When I read it I can hear Greg's voice, and his personality, very clearly, and it is always calm, but enthusiastic, with a touch of humour. Read it yourself and enjoy - coding with Spring is still fun after all these years!

— **Dr. Dave Syer**

*Senior Consulting Engineer and co-founder of Spring Boot*

# ABOUT THE AUTHOR

---

**Greg L. Turnquist** works on the Spring team as a principal developer at VMware. He is a committer to Spring HATEOAS, Spring Data, Spring Boot, R2DBC, and Spring Session for MongoDB. He wrote *Hacking with Spring Boot 2.3: Reactive Edition* as well as Packt's best-selling title, *Learning Spring Boot 2.0 2nd Edition*. He co-founded the Nashville Java User Group in 2010 and hasn't met a Java app (yet) that he doesn't like.

Be sure to subscribe to *Spring Boot Learning*, the YouTube channel where you learn about Spring Boot and having fun doing it at **[YouTube.com/SpringBootLearning](https://www.youtube.com/SpringBootLearning)**.

# PREFACE

---

# WHAT THIS BOOK COVERS

---

*Chapter 1, Building a Web App with Spring Boot* - Learn how to build web applications using Spring MVC.

*Chapter 2, Data Access with Spring Boot* - Access data stores using Spring Data.

*Chapter 3, Developer Tools for Spring Boot* - Enhance your developer experience with all the tools afforded by Spring Boot.

*Chapter 4, Testing with Spring Boot* - Get a hold of testing tools and how Spring Boot empowers testing your applications.

*Chapter 5, Operations with Spring Boot* - See how to manage your applications after they go to production.

*Chapter 6, Building APIs with Spring Boot* - Build JSON-based APIs using different tactics and tools from the Spring portfolio.

*Chapter 7, Messaging with Spring Boot* - Create asynchronous, message-based solutions using Spring Boot and Testcontainers.

*Chapter 8, Securing your Application with Spring Boot* - Learn how secure your application with the most powerful tools available.

# WHAT YOU NEED FOR THIS BOOK

---

Spring Boot 2.4 supports Java 8 and higher. This book is written using Java 8. Use [sdkman](#) to install, manage, and even switch between different distributions and versions of Java.

Spring Boot 2.4 is able to bake [Docker](#) containers. There is also Docker-based testing support through 3rd party tools like [Testcontainers](#). For those sections, you'll need to install Docker.

If you use Mac, you should consider [Homebrew](#) as a package manager for certain utilities.

You need either an IDE or a good editor. Recommended options include:

- IDE
  - [IntelliJ IDEA](#)
  - [Spring Tool Suite](#)
  - [VS Code](#)
  - [Eclipse](#)
- Editor

- [Sublime Text 3](#)

- [Atom](#)

# WHO SHOULD READ THIS BOOK

---

This book is to help developers new to Spring Boot as well as experienced Spring developers.

It shows how to get operational, fast, with some of the best coding practices available.

It helps the reader focus on adding business value to their applications and not get distracted with infrastructure.

# OPTIMAL VIEWING

---

Adjust your e-reader to use a smaller font. This book is loaded with code examples. Attempting to show things too big will cause unneeded wrapping, culminating in a less than desirable experience.

# CONVENTIONS

---

In this book, you will find a number of styles of text.

Code found in the text are shown like this: "A Spring MVC controller is flagged with an `@Controller` annotation."

A block of code looks like this,  
Example block of code

```
@RestController
class WebController {
    @GetMapping("/api")
    String home() {
        return "Hello, world!";
    }
}
```

When certain parts of the code are described in more detail, they are annotated with a circled number at the end of the line.  
Code with annotated lines

```
@Component ①
public class RepositoryDatabaseLoader {

    @Bean ②
    CommandLineRunner initialize(ItemRepository repository) { ③
```

```
    return args -> { ④
        repository.save(new Item("Alf alarm clock", 19.99));
        repository.save(new Item("Smurf TV tray", 24.99));
    };
}
```

① This comment describes the line above with the (1) comment.

② And this is for the (2) line.

③ The (3) line.

④ The lambda expression with (4).

Sometimes you'll see chunks of code that have comments at the end with no numbering. To improve readability in the code and in the manuscript, line breaks are sometimes forced.



Warnings appear like this.



Tips appear like this.



Notes appear like this.

Important facts appear like this.

# READER FEEDBACK

---

The most valuable feedback you can leave is an honest review.

Please visit your book provider when you finish and [share your personal opinion of \*Hacking with Spring Boot 2.4: Classic Edition\*.](#)

# SUPPORT

---

If you have issues with getting your copy, contact the provider.

If you are having issues with the code, please file a ticket at  
<https://github.com/hacking-with-spring-boot/hacking-with-spring-boot-classic-code>.

If there is an issue with the manuscript, please email me at  
[greg@greglturnquist.com](mailto:greg@greglturnquist.com).

# DOWNLOADING THE CODE

---

You can download the example code from GitHub at  
<https://github.com/hacking-with-spring-boot/hacking-with-spring-boot-classic-code>, free of charge.

Are you ready? *Chapter 1, Building a Web App with Spring Boot* is waiting!

1

---

# BUILDING A WEB APP WITH SPRING BOOT

---

Working with Spring Boot is like pair-programming with the Spring developers.

~ Josh Long @starbuxman

Seven years ago, something amazing happened. Spring Boot 1.0 was released. On April 1st, 2014, project lead Phil Webb published a blog article detailing the first stable release.<sup>[1]</sup>

And the crowd went wild. The Java community embraced this amazing culmination of engineering and creative art with fervent excitement. Searching Twitter for #springboot tweets generated an avalanche of activity.

Three years later, in 2017, the marketing department of Pivotal tweeted that Spring Boot's download growth had achieved 19.7MM downloads *per month*.<sup>[2]</sup>

Something was done. And it was done right.

In this release of *Hacking with Spring Boot 2.4: Classic Edition* (a follow-up to *Hacking with Spring Boot 2.3: Reactive Edition*), you are going to dive head first into all kinds of goodness. You'll build some features for a shopping cart-e-commerce system. Then you'll accelerate things chapter by chapter, taking a new angle on building your application. And you'll use Spring Boot to make it happen.

You'll explore Spring Boot's powerful tools to speed up development efforts as well as production-ready, cloud-native features.

If you've never used Spring Boot before, get ready for some fun. This book is jam-packed with extra goodies that perhaps you weren't aware of. Your knowledge will be extended so you can take full advantage of its features.

## What is Spring Boot?

Maybe you've seen a Spring Boot presentation at a conference or a JUG meeting? Perhaps you've encountered a quick-fire demo.

What, exactly, is it?

Spring Boot is a fast, opinionated, portable, and production-ready assembly of the Spring portfolio.

- **Fast** - By making decisions based on many factors including your dependencies, Spring Boot helps you build your app quickly.
- **Opinionated** - Spring Boot makes assumptions based on what it sees. These opinions can easily be overruled when needed. But based on feedback, these pre-made opinions have served the community well.
- **Portable** - Built on top of Java's de facto standard toolkit, the Spring Framework, Spring Boot apps can be run *anywhere* a JDK can be found. No need for a certified application server or other vendor-specific product. Build your app, package it up using Boot's tools, and you're ready to deploy!
- **Production Ready** - Make no mistake, Spring Boot isn't vaporware. And it's not confined to tiny stuff (but it's *great* for micro/macro/anysize services!) Spring Boot is real and widely adopted. As an example, check out [this blog article from Netflix](#), one of the largest Java shops out there.



You can also [subscribe to my YouTube channel](#) and see my fun videos about Spring Boot.

Using this powerful and widely adopted stack, you'll build a system with speed AND stability.

In *Hacking with Spring Boot 2.4: Classic Edition* you will explore the tried and true paradigms that made Spring the *de facto* toolkit for Java development.

In this chapter you will cover the following topics:

- Launching an e-commerce platform using <https://start.spring.io>.
- Exploring Spring Boot's management of third-party libraries.
- Running your app inside your IDE with no standalone container.
- Building a web layer using Spring MVC.

## Say hello to Spring MVC

Web applications dominate the landscape. Sure, you can find a thick client here and there. But seriously, when was the last time you met a Swing or Griffon developer?

And when it comes to building web applications, the most popular toolkit has been Spring MVC.<sup>[3]</sup>

With new startups needing to get to market *fast*, Spring MVC is quick, easy, and perfectly suited to getting things running. And when your needs grow, Spring MVC can scale with you.

If you're completely new to the Spring portfolio, never fear. Spring MVC isn't hard. In fact, it's pretty down-to-earth.

Spring MVC is built atop Java's well-honed servlet API. This is part of Spring's magic. While it leverages well established APIs, it seeks to reduce Java complexity. You don't have to cobble together a servlet. Instead, define your controller methods and Spring Boot will hook it up to your web container of choice through a standardized servlet. You can keep your focus on solving business problems while Spring Boot focuses on solving infrastructure problems.

And that is how you are going to develop e-commerce features throughout this book—using Spring MVC along with other parts of the Spring portfolio.

## Building an E-Commerce Platform with Spring Boot

So you're all fired up to build an e-commerce platform using Spring MVC? Great. To get going on any new project, you should do what everyone else does. Go to your favorite search engine and type "spring boot build file."

Just kidding!

You actually *don't* have to rummage around on a search engine or StackOverflow to get off the ground. There's a website geared toward getting you started. The **Spring Initializr** (yes, that's how you spell it) at <https://start.spring.io> lets you enter the

barebones details about your project, pick your favorite build system, the version of Spring Boot you want, and last but not least, select all the dependencies you need. Then a ZIP file is generated containing your app, ready to go!

## Spring Initializr

The screenshot shows the Spring Initializr interface on a Mac OS X browser window. The URL in the address bar is start.spring.io. The page itself has a light gray background with various configuration sections:

- Project**: A section with two radio buttons: "Maven Project" (selected) and "Gradle Project".
- Language**: A section with three radio buttons: "Java" (selected), "Kotlin", and "Groovy".
- Spring Boot**: A section with five radio buttons for Spring Boot versions: "2.5.0 (SNAPSHOT)", "2.5.0 (M2)", "2.4.5 (SNAPSHOT)" (selected), "2.4.4" (selected), and "2.3.10 (SNAPSHOT)".
- Project Metadata**: A section with three buttons: "GENERATE", "EXPLORE", and "SHARE...".

You can start by picking “Maven Project.” Then, select “2.4.4” as the version of Spring Boot you wish to use.



While Spring Boot supports both Maven and Gradle as first-class citizens, this book is built using Maven.

Under the section labeled "Project Metadata", enter the following details:

- **Group** - com.gregturnquist
- **Artifact** - hacking-spring-boot-ch1-classic

Now for the real kicker—picking what you want your application to do. For example, type "Web" in the search box, and you'll see several things pop up: "Spring Web", "Spring Reactive Web", and some other technologies we aren't exploring in this chapter.

- **Spring Web** - traditional, imperative-based Spring MVC web stack
- **Spring Reactive Web** - Reactive Streams-based Spring WebFlux web stack



This is enough to get things moving. However, you are free to add more metadata or sift through the stack of supported features. Simply click on the text "ADD DEPENDENCIES" and scroll through the options.

Fill in the following details:

- Name - Hacking with Spring Boot - Chapter 1 - Classic
- Description - Demo project for Hacking with Spring Boot
- Package Name  
`com.gregturnquist.hackingspringboot.classic`



This site has lots of overrides. You can switch from building a JAR to a WAR. You can opt for either Java 8 or the newest stable release, Java 11, or even bleeding-edge Java 15. This book will focus on Java 8, and unless you have an explicit requirement such as deploying to an older container, it's best to "make JAR not WAR."

To build your e-commerce platform, clear out the selected modules and select the following items:

- Spring Web (Spring MVC + embedded Apache Tomcat)
- Thymeleaf (template engine)

With these items selected, click on “Generate Project”.



Many tools leverage the Spring Initializr site. IntelliJ IDEA and the Spring Tool Suite both make it possible to create new projects inside the IDE. You have access to the same options shown above. They invoke the web site's REST API and imports your new project. You can even use cURL or HTTPie on the command line!

So get to it and unpack that tasty ZIP file. What have you got?

- A `pom.xml` build file with a Maven wrapper (no need to install Maven!)
- A `HackingSpringBootApplication.java` application class
- An `application.properties` file
- A `HackingSpringBootApplicationTests.java` test class

You built an empty Spring Boot project. Now what?

Before you sink our teeth into writing your first Spring MVC controller, take a peek at the build file. It's quite terse, but carries some key bits.

## Project Parent

You'll find the parent at the top of your Maven build file.  
Parent details in your build file

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

```
<version>2.4.4</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
```

This clause designates `spring-boot-starter-parent` as the basis for your project, specifically version 2.4.4.

What exactly does this give you?

By adopting Spring Boot's starter parent, you inherit a swath of predefined properties, dependencies, and plugins. One of the biggest is Spring Boot's predefined list of dependencies. This includes the entire Spring portfolio along with many supported 3rd-party libraries such as Jackson, Apache Tomcat, and more. Need a particular library? Simply add it to the build and let Spring Boot select the version.

As a bonus, anytime Spring Boot comes out with a new release, you only have to update one version. One! Everything will move up, and you'll know it's supported.

## Application Metadata

Next is the metadata about your application.

Application metadata

```
<groupId>com.gregturnquist</groupId>
<artifactId>hacking-spring-boot-ch1-classic</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>Hacking with Spring Boot - Chapter 1 - Classic Web</name>
<description>Demo project for Hacking with Spring Boot</description>
```

The **group**, **artifact**, **name**, and **description** are exactly as you entered them on <https://start.spring.io>. The version number starts at `0.0.1-SNAPSHOT`.

## Spring Boot Starters

One of Spring Boot's most amazing features are its **starters**. These are published artifacts that contain no code, but simply bring in key libraries. They are modular and strategically designed to grab exactly what your application needs.

Looking below, you can see exactly what you picked on the website.

### Project dependencies

```
<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-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

The first two dependencies in this clause include:

- `spring-boot-starter-thymeleaf` - Thymeleaf templating engine
- `spring-boot-starter-web` - Spring MVC web stack

This matches the fact that you selected Spring Web and Thymeleaf.

But it's important to note that one extra dependency was added automatically: `spring-boot-starter-test`. In this day and age, testing is so critical that the Spring Initializr includes this test-scoped tool unconditionally.

Spring Boot's Test starter brings in:

- AssertJ
- Hamcrest
- HTMLUnit
- JSONassert
- JsonPath
- JUnit 5
- Mockito
- Selenium
- Spring Test
- XMLUnit

With the most popular Java testing tools at your fingertips, there's no reason to NOT delve into crafting tests. See *Chapter 4, Testing with Spring Boot* for more details!

## Spring Boot Maven Plugin

For it all to come together, you need one extra thing. And the Spring Initializr-generated `pom.xml` file has it.

### Spring Boot Maven plugin

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

The **Spring Boot Maven Plugin** will build your Java application and turn it into an executable fat JAR, otherwise known as an über JAR.

What issue does this plugin solve, exactly?

The Java JAR specification does NOT cover embedding JAR files inside other JAR files. This plugin grabs all of your declared dependencies and inserts them into the final JAR file. It also includes some glue code to load classes when you launch the app.

There is additional support to turn the JAR file into an executable that even honors startup init run levels.



Alternative plugins that unpack 3rd-party class files and insert them into your target JAR, a practice known as "shading", can be haphazard and possibly infringe upon a library's license.

## Initial Code

In addition to the build file you just peeked at, the Spring Initializr provides the following application file.

### Spring Boot Application Code

```
@SpringBootApplication
public class HackingSpringBootApplication {

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

This class contains:

- `@SpringBootApplication` - a composite annotation that brings in **autoconfiguration** and **component scanning**.
- `main(String[] args)` - a runnable function to launch your application.

- `SpringApplication.run(HackingSpringBootApplication.class, args)` - a Spring Boot hook to register this class as the launch point for your application.

This class contains the magic that says "I no longer need to install my app in a web container!"

How?

Through the combination of **autoconfiguration** and **component scanning**.

## Autoconfiguration

Spring Boot has something called **autoconfiguration**. Autoconfiguration is conditional logic that looks at various settings and based upon what's found, activates various beans.

Spring Boot has a multitude of autoconfiguration "policies" that look at:

- The classpath
- Various properties
- The existence of certain beans
- The lack of the existence of certain beans
- More...

These various aspects of your application are using to make deductions and activate various components. For example, there is one policy of keen value later in the chapter called `WebMvcAutoConfiguration` that only activates if:

1. You have a servlet-based container.
2. You have Spring MVC on the classpath.
3. No one has declared a bean of type `WebMvcConfigurationSupport`.

The servlet container that gets activated in this scenario is **Apache Tomcat**. If you don't know the history of Spring, Apache Tomcat has been a go-to servlet container for years. It is lightweight, yet resilient and able to serve almost all your needs. Later in this book, though, you'll learn how you can easily swap it out for just about any other servlet container.

Spring MVC is Spring's web stack built on top of standarized servlet containers. (which we'll delve into shortly).



When you have `spring-boot-starter-web` in your build file, it brings in `spring-boot-starter-tomcat`, which activates Apache Tomcat *automatically*.

`WebMvcConfigurationSupport` can be deemed an "escape hatch." If no such bean exists, Spring Boot will create the beans needed to

setup Spring MVC. However, if this doesn't fit your needs, you can create your own instance with all the settings you wish, and fashion it into a bean. It will cause the autoconfiguration policy to back off and not run.

And that's one of Spring Boot's coolest features. Beans that are enabled due to one set of criteria, but will back off and NOT take effect based on another set of criteria.

## Component Scanning

There are two modes of registering beans in a Spring application. You either declare every bean yourself in a configuration class, apart from where the class itself is defined. Or you flag their existence and let Spring do what is known as **component scanning**. In the latter situation, you let Spring search for and register the beans automatically.

When a Spring application starts up, all these beans are registered in an **application context**. If you are already familiar with Inversion of Control and Dependency Injection, free free to skip along to the next section.

Dependency injection was a radical concept back in the early 2000s. At the time, the concept of registering services with an address and letting collaborating components "look them up" using the Service Locator pattern was most prevalent.

This approach was solid, but led to very static applications. Being able to swap out a real database service with a mock was very difficult. At least, not without the collaborator being aware of it.

Rod Johnson blazed the trail with a book that illustrated a much more nimble way of coding. Soon after, the code from his book was picked up by Juergen Hoeller and several others and was turned into what is today known as the Spring Framework, resulting in their 1.0 final release on March 24th, 2004.

The uptake for the Spring Framework was so powerful and its pursuit of reducing Java complexity was so strong, that some present-day surveys have shown one out of every two Java developers use some aspect of the Spring portfolio.

## Creating a Spring MVC Controller

Now that you have taken an initial tour through a bare Spring Boot project, it's time to get down to business and code your first controller.

You can start by writing the following Java class.

Coding your Spring MVC controller

```
package com.greglturnquist.hackingspringboot.classic;

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

import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ServerController {

    private final KitchenService kitchen;

    public ServerController(KitchenService kitchen) {
        this.kitchen = kitchen;
    }

    @GetMapping("/server")
    List<Dish> serveDishes() {
        return this.kitchen.getDishes();
    }
}
```

This is a simple controller that does a few things.

- `@RestController` is a Spring web annotation that marks this class as a controller that doesn't use any templating. Instead, it responds to web calls by serializing results and writing them straight into the HTTP response body.
- `KitchenService` (which you'll write shortly) is provided to this controller via **constructor injection**. When the app starts, Spring will seek out an instance of this service and feed it automatically to the constructor.
- `@GetMapping(...)` is another Spring web annotation that routes HTTP GET /server calls to the `serveDishes` method.

List<Dish> is a type that will return back a collection of prepared meals, serialized into JSON.

So what about that needed KitchenService? Write some code like this:

### Coding a real kitchen service

```
package com.greglturnquist.hackingspringboot.classic;

import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.stream.Stream;

import org.springframework.stereotype.Service;

@Service
public class KitchenService {

    /**
     * Generates a list of random dish(es).
     */
    List<Dish> getDishes() {
        return Arrays.asList(randomDish());
    }

    /**
     * Randomly pick the next dish.
     */
    private Dish randomDish() {
        return this.menu.get(picker.nextInt(3));
    }

    private List<Dish> menu = Arrays.asList( //
        new Dish("Sesame chicken"), //
```

```
    new Dish("Lo mein noodles, plain"), //  
    new Dish("Sweet & sour beef"));  
  
    private Random picker = new Random();  
}
```

Whoah. What's all that? Let's take it apart.

- `@Service` is an annotation that marks this as a Spring bean. It will automatically be picked up and created by Spring Boot during **component scanning** and then injected into `ServerController` as needed.
- `getDishes()` has the same method signature you used earlier, only instead of providing three dishes, it generates a random dish from the kitchen every 250ms.

The earlier simulation was a fixed list, so calling it a simulation was kind of weak. *This* example is a little more lifelike because it uses `Stream.generate()` to initiate an infinite stream of prepared dishes.

Peeking inside the `generate()` function, you can see a Java lambda function, `() → randomDish()`. This Java 8 lambda function lets you invoke pretty much anything to generate entries in the Java 8 `Stream`. In this case, you have `randomDish()` to spruce up the simulation.



All of these Stream methods are great, but it's important to realize that Java's Stream API isn't completely lazy. With servlet-based programming, you have to resort to something more complex, like Server-Sent Events, WebSockets, or crafting an asynchronous response using StreamingResponseBody or other. These methods require proper balancing with server-side database transactions and whatever else is being orchestrated. If you find yourself needing this a lot, you should entertain switching to Spring WebFlux (and visiting [\*Hacking with Spring Boot 2.3: Reactive Edition\*](#)).

The last bits of KitchenService use an instance of Java Random to pick a random integer, and look up the corresponding menu item.



If this randomly-generated dish-picking algorithm sounds a bit weird, relax. As you work on your e-commerce platform throughout this book, you'll migrate away from generated simulations and move toward filling out a real shopping cart. But since this chapter's focus is on building a classic web layer, you need this simulation for now.

To put in the final piece of this example-turned-into-code, write the following Dish class.

Defining a Dish domain object

```
package com.greglturnquist.hackingspringboot.classic;

class Dish {

    private String description;
    private boolean delivered = false;

    public static Dish deliver(Dish dish) {
        Dish deliveredDish = new Dish(dish.description);
        deliveredDish.delivered = true;
        return deliveredDish;
    }

    Dish(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public boolean isDelivered() {
        return delivered;
    }

    @Override
    public String toString() {
        return "Dish{" + //
            "description='" + description + '\'' + //
            ", delivered=" + delivered + //
            '}';
    }
}
```

This simple POJO has a `description` field and a `delivered` field. Additionally it has typical getters and setters along with a `toString` method. A suitable `equals` and `hashCode` methods are also desirable, but not necessary for this example.

With all this in place, you can now run your application.

As mentioned earlier, by using embedded Apache Tomcat, there's no need to install a web container. Instead, you can right-click `HackingSpringBootApplication` in your IDE and run it, or you can type this on the command line.

```
$ ./mvnw clean spring-boot:run
```

Either way, you should see something similar to the following on the console.

```
2020-02-16 17:34:20.778 : Starting HackingSpringBootApplication on  
retina with PID 67100 (/Users/gturnquist/personal/hacking-spring-  
boot-code/1-classic/target/classes started by gturnquist in  
/Users/gturnquist/personal/hacking-spring-boot-code/1-classic)  
2020-02-16 17:34:20.783 : No active profile set, falling back to  
default profiles: default
```

```
2020-02-16 17:34:22.236 : Tomcat started on port(s): 8080  
2020-02-16 17:34:22.240 : Started HackingSpringBootApplication in  
1.767 seconds (JVM running for 6.064)
```

From there, you can use any web CLI tool you want. To query the service using cURL (<https://curl.haxx.se/>) such that it doesn't wait for the *entire thing to finish*, type this.

```
$ curl -v localhost:8080/server
```

You should then see the following.

```
* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /server HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK

data:{"description":"Sesame chicken","delivered":false}

data:{"description":"Sesame chicken","delivered":false}

data:{"description":"Sweet & sour beef","delivered":false}

data:{"description":"Lo mein noodles, plain","delivered":false}

data:{"description":"Sweet & sour beef","delivered":false}
```

Ta-dah. You did it. You built a web service without getting knee-deep in infrastructure. All you had to do was define a

controller, a supplying service, and a domain object, and you were up and running.

That is the power of Spring Boot.

No need to register view resolvers, web method handlers, and other infrastructure beans. That's provided by Spring Boot.



Instead of wasting time building these same components again and again for every app, Spring Boot pre-bakes them for you. You keep your focus on connecting routes to services. And that way, you can focus on delivering value to your customers.

From here, you can make more revisions. For example, in the conceptual phase, you saw the concept of the server changing the state of a dish using a mapping function.

You'd do that by adding another web method to `ServerController`.

**Delivering dishes through a function**

```
@GetMapping("/served-dishes")
List<Dish> deliverDishes() {
    return this.kitchen.getDishes().stream() //
        .map(dish -> Dish.deliver(dish)) //
        .collect(Collectors.toList());
}
```

- This method has a different route, `/served-dishes`.

- It calls the same KitchenService.getDishes() function.
- However, it takes the List<Dish> result and maps one over a new function, deliver(), as it arrives.
- The deliver() function takes the dish and produces a *new* Dish, with its delivered field set to true.



When doing functional programming, it's preferable to avoid mutating state, and instead produce new objects.

If you re-start your app, and cURL the results, you can expect to see this.

```
$ curl -v localhost:8080/served-dishes

*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /served-dishes HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
<
data:{"description":"Sweet & sour beef","delivered":true}

data:{"description":"Sesame chicken","delivered":true}

data:{"description":"Sweet & sour beef","delivered":true}
```

```
data:{"description":"Sesame chicken","delivered":true}
```

This is almost the same, except that every dish now reports "delivered":true.

Given all this, it's not hard to imagine doing other transformations. Taking a set of data provided by the KitchenServer, you can bend it, transform it, and manipulate it as needed for the consumer of the service.



While it's easy to tweak things at any phase, Phil Webb, the lead for Spring Boot recommends keeping web controllers light. Avoid weighing them down with business logic. Instead keep your controllers focused as the layer for connecting web requests to service methods. For GET calls, this is where data gets serialized. For PUT, POST and other incoming operations, the controller is where inputs are extracted and passed along to the underlying services responsible for handling.

## Tiptoeing into Templates

So far, you've built a web controller that generates JSON. That's a pretty nifty thing for just getting warmed up on Spring's web tech.

But that is not all that's needed to build web services. Frequently, you need to serve web pages. And so you should

turn to templating libraries. When it comes to web programming, Thymeleaf (<https://www.thymeleaf.org/>) is 100% HTML-compliant. It also is future-proofed in the sense that if you switch this project (or another one) to reactive, Thymeleaf fully supports **Reactive Streams**.

Check out the barebones example shown below.

Crafting a simple home controller

```
package com.greglturnquist.hackingspringboot.classic;

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

@Controller
public class HomeController {

    @GetMapping
    String home() {
        return "home";
    }
}
```

Not too much to pick apart:

- `@Controller` indicates this is a Spring web controller, but based upon templates instead of raw JSON or XML.
- `@GetMapping` maps HTTP GET requests. Because neither it nor the class itself has a path, it defaults to `/`.
- The return type is `String` indicating it will return the name of a template.

- The body of the method just returns a bare string, `home`.

This is the simplest web controller you can write. It handles GET / web calls coming in to Apache Tomcat. Spring MVC fields the request and routes it to this method. The `home` method does, really, nothing but hand back the name of a template.

Spring Boot autoconfigures a Thymeleaf view resolver to transform the value of `home` into `classpath:/templates/home.html` (notice the prefix, `classpath:/templates/` and the suffix, `.html`). Using that, write the following HTML web page in `src/main/resources/templates/home.html`.

### Writing a simple Thymeleaf template

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" />
    <title>Hacking with Spring Boot - Getting Started</title>
</head>
<body>
    <h1>Welcome to Hacking with Spring Boot!</h1>

    <p>
        Over the span of this book, you'll build
        different parts of an e-commerce system
        which will include fleshing out this web
        template with dynamic content, using the
        power of Spring Boot.
    </p>

</body>
</html>
```

---

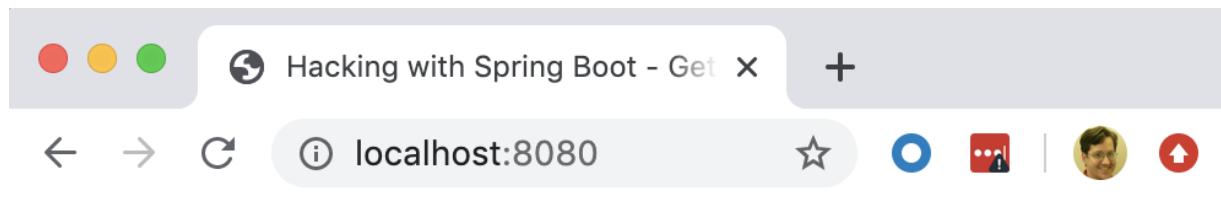
Nothing fancy. It's just a web page describing what you are going to do over the life of this book.

As you embrace new features and aspects of Spring Boot throughout the book, you'll augment this site to make it dynamic and vibrant. But for now, start with a fixed, static page.



Thymeleaf has a DOM-based parser. That means that all HTML tags must be closed. I know that certain HTML5 elements are valid without a closing tag, e.g. `<img>`. But if you put that in a Thymeleaf template without either `</img>` or simply as `<img />`, it will generate a runtime error.

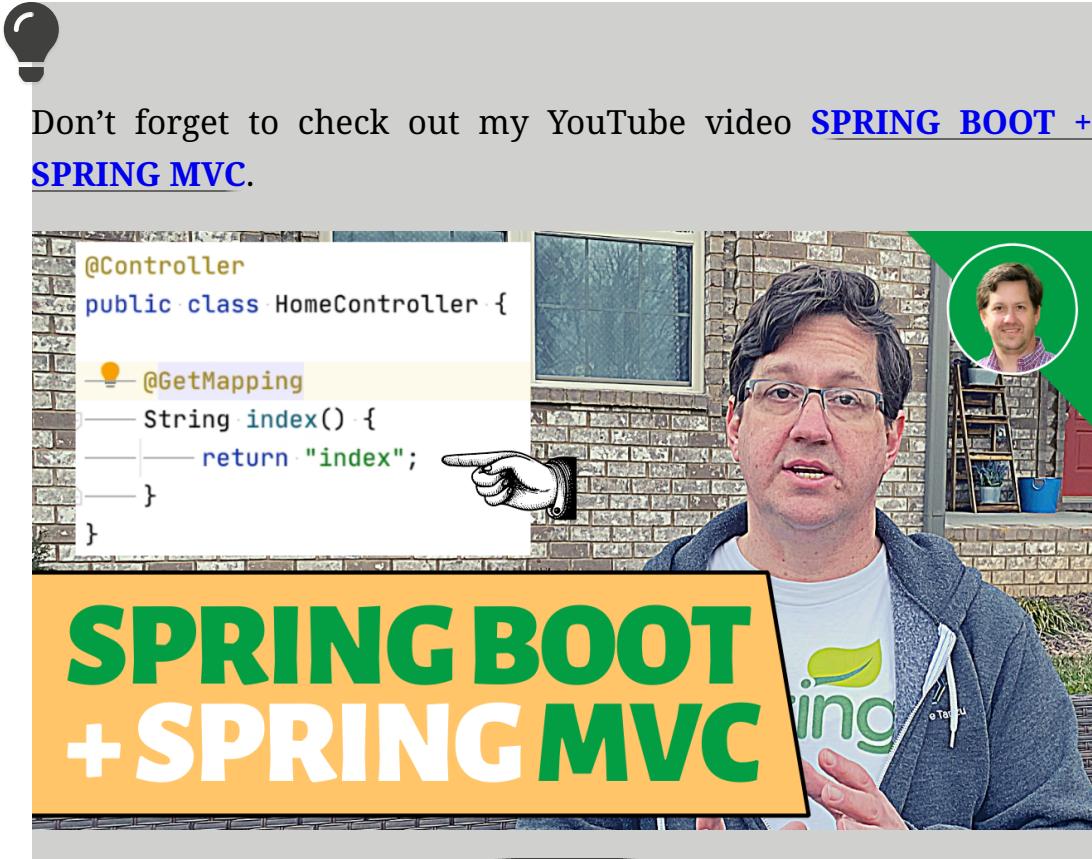
Run the application, either in your IDE or by typing `./mvnw spring-boot:run` on the terminal. Visit <http://localhost:8080>, and you should see the template rendered in your browser.  
Thymeleaf template rendered as HTML



# Welcome to Hacking with Spring Boot!

Over the span of this book, you'll build different parts of an e-commerce system which will include fleshing out this web template with dynamic content, using the power of Spring Boot.

Congrats. With this simplistic template you have managed to render HTML in the browser. Now stay tuned as you continue to enhance things throughout this book.



## Summary

In this chapter you:

- Took a first glance at Spring MVC and web programming by crafting a controller and a service.
- Ran your first Spring Boot application and consumed this set of dishes using cURL.
- Crafted your first Thymeleaf template that served a static web page.

In the next chapter, *Data Access with Spring Boot*, you will dive into accessing data using Spring Data, relational databases, and how to both consume and supply data!

[1 Spring Boot 1.0 GA Released](https://spring.io/blog/2014/04/01/spring-boot-1-0-ga-released), Phil Webb,

[2](#) The Spring team was spun off as part of Pivotal Software in 2012, and reacquired by VMware in 2019.

[3](#) Per the 2017 survey conducted by *Zeroturnaround* (now JRebel by Perforce), Spring MVC is almost double the #2 web toolkit, JSF.  
<https://www.jrebel.com/blog/java-web-frameworks-index>

2

---

# DATA ACCESS WITH SPRING BOOT

---

With Boot you deploy everywhere you can find a JVM basically.

~ Oliver Drotbohm @odrotbohm

Nobody builds an application without data.

It's as simple as that. Maybe today you aren't integrating with that 3rd-party service. Maybe today you aren't securing your pages and API. Maybe today you aren't connecting to a message broker.

But if you haven't hooked up to a database to store and retrieve data, you don't have a real demo.

Data is the lifeblood of applications, and Spring Boot's top-notch support for Spring Data makes data management not just achievable but *fun*.

In the previous chapter, you learned some fundamentals of web programming with Spring MVC. Then you used that knowledge to create a web controller that could send JSON data as well as serve up a simple HTML web page.

In this chapter, you are going to:

- Define your e-commerce application's domain object.
- Create a repository to store and retrieve objects.
- Wrap it inside a service.

So let's go!

## Defining Your E-Commerce App's Domain

In the previous chapter, you dug into the concept of fielding requests by thinking about a restaurant server ferrying requests between customers and the kitchen. You turned that into code with a simulated source of data.

In this chapter, now that you're embracing a real database, you can actually start modeling your e-commerce site. Before you do that, you need to add some new dependencies to your `pom.xml` build file.

JPA dependencies added to your `pom.xml` build file

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

If you're catching up and don't have a project to add this to, just use [start.spring.io](https://start.spring.io) with the following options:

- Maven Project
- Java 8
- Spring Boot 2.4.4
- Group: com.greglturnquist
- Artifact: hacking-spring-boot-ch2-classic
- Name: Hacking with Spring Boot Ch. 2 - Classic Data
- Description: Demo project for Hacking with Spring Boot
- Package: com.greglturnquist.hackingspringboot.classic
- Dependencies:
  - Spring Data JPA

- H2
- Thymeleaf
- Spring Web

Apply these settings and click Generate to get a ZIP file. After you download the project and unpack it.



Or, you can just view my already-written code at <https://github.com/hacking-with-spring-boot/hacking-with-spring-boot-cde/tree/master/2-classic>.

The first dependency, `spring-boot-starter-data-jpa`, is one of Spring Boot's **starters**. If you look at this starter's `pom.xml` file (found at <https://github.com/spring-projects/spring-boot>), you can see its dependencies.

*Table 1. Datastore Modules*

<b>Module</b>	<b>Description</b>
<code>spring-boot-starter-aop</code>	Spring Framework's AOP module, used for various interceptors used for transactions and other operations.

<b>Module</b>	<b>Description</b>
spring-boot-starter-jdbc	Spring Framework's JDBC module, used for other key relation data operations. Also brings in the data sources used by JPA.
jakarta.transaction-api	Jakarta EE's Transaction API (JTA).
jakarta.persistence-api	Jakarta EE's Persistence API (JPA).
hibernate-core	Hibernate, the implementation of JPA.
spring-data-jpa	Spring Data JPA itself.

That starter brings in Spring Data JPA.

The second dependency added to your build file is com.h2database.h2, an embedded relational database. Often used for test purposes, you can use it as your principle data store for the early phases of application design.

With these additions to your build file, you can noodle out an inventory of products placed in a shopping cart. To do that, you need to model several things.

*Table 2. E-Commerce Domain Definitions*

<b>Domain Object</b>	<b>Description</b>

Domain Object	Description
Inventory Item	Needs some sort of serial number, a price, and a description.
Cart	Needs a unique identifier along with a list of items.
Item in a Cart	Needs an inventory item and quantity of that item in the cart.

There are whole books dedicated to entity modeling, domain-driven design, and more. No need to do a full analysis here. Instead, let's jump ahead and define an **Inventory Item**.

### Data representation of an inventory Item

```

@Entity
public class Item {

    private @Id @GeneratedValue Integer id;
    private String name;
    private double price;

    protected Item() {}

    Item(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // getters, setters, equals, and hashCode
    // omitted for brevity
}

```

Okay, this doesn't look too hard. So let's take it apart.

- id, name, and price are the attributes you need as defined in the table above.
- A constructor with the data fields is crafted, deliberately excluding the id field.

Since you are using Spring Data JPA, you need to mark which field is the primary key. You do so by using @Id, a JPA annotation.



Spring Data has custom solutions for every data store. But some concepts are captured in Spring Data Commons, at least where it makes sense. For example, almost every data store needs the concept of a "serial number." In relational databases, this is called a "primary key." When using Spring Data JPA, you have JPA's javax.persistence.Id. For all other supported data stores, they can use Spring Data Commons' @Id annotation.

At this stage, you are defining the POJO class that will be used by JPA itself. So any JPA annotations are permitted. In fact, you are encouraged to follow JPA's recommended practices for defining objects to store.

When it comes to interacting with data stores, Spring Data encourages you to use proper practices for that store. There is

no one-size-fits-all, so don't presume that what works for Redis also works for JPA.



Avoid designing domain objects to be stored in multiple database engines. A POJO that works for a relational data store using JPA will NOT fit well in Apache Cassandra and vice versa. There are use cases for interacting with more than one data store, but they are the exception. For better details on designing for the database you have in mind, I'd recommend finding a more specific book.

To continue designing your e-commerce domain, you need to capture the idea of adding an Item to a cart. For that, you need something like this.

Data representation of a quantity of Item objects

```
@Entity  
class CartItem {  
  
    private @Id @GeneratedValue Integer id;  
    private @ManyToOne(fetch = FetchType.LAZY) Cart cart;  
    private @ManyToOne(fetch = FetchType.LAZY) Item item;  
    private int quantity;  
  
    protected CartItem() {}  
  
    CartItem(Item item, Cart cart) {  
        this.item = item;  
        this.cart = cart;  
        this.quantity = 1;  
    }  
}
```

```
// getters, setters, equals, and hashCode  
// omitted for brevity  
}
```

The final piece for an e-commerce system is to have a collection of these `CartItem` objects.

Data representation of a Cart of goods

```
@Entity  
class Cart {  
  
    private @Id String id;  
    private @OneToMany(mappedBy = "cart", cascade = CascadeType.ALL,  
    orphanRemoval = true) List<CartItem> cartItems;  
  
    protected Cart() {}  
  
    public Cart(String id) {  
        this(id, new ArrayList<>());  
    }  
  
    public Cart(String id, List<CartItem> cartItems) {  
        this.id = id;  
        this.cartItems = cartItems;  
    }  
  
    // getters, setters, equals, and hashCode  
    // omitted for brevity  
}
```

This `Cart` object also has a unique identifier as well as a Java Collection of `CartItem` objects. It includes a private, empty

constructor as well as one to store everything.



Right now, you don't (yet) have the concept of a user. You can't say "find Alice's cart" or "delete Bob's cart." Don't panic! You'll get to user management later in this book and alter your domain model to work with that.

For now, let's focus on building an inventory and putting them in a cart.

## Creating a Repository

In the previous section, you defined your first round of domain objects for storage in H2. What you haven't delved into, yet, is exactly how Spring Data is going to help you read and write to H2.

People have explored standardizing SQL stores, but the effort has been fraught with challenges. Every SQL engine has slight differences. The SQL standard itself has gaps. In fact, SQL isn't confined to a single standard! And the task of storing objects into SQL stores is what brought on the attempts to standardize with JPA, which itself has its own quirks and issues as you move between data stores.

And that is not very encouraging.

So what does Spring Data do to leap over this problem?

One of Spring's powerful paradigms is usage of the **template pattern**, e.g. `JdbcTemplate`, `RestTemplate`, `JmsTemplate`, and more. The strength of these various tools is capturing operations in a typesafe way and abstracting away verbose cruft, ensuring interactions are performed correctly.

The simplest example is how `JdbcTemplate` takes away the developer's need to open and close database connections.



My gateway into the Spring Framework, long ago, was the discovery of `JdbcTemplate`. I had built an app on another platform with over two hundred SQL statements. Connection and cursor management was *hard*. Being able to delegate such fussy management to a framework was awesome!

Spring Data provides a custom template for each data store, including `HibernateTemplate`. Honing in on one particular data store, there are operations that are store-specific, meaning that you can use the full richness of the database to do whatever you need.

But that is not all. There is another layer offered with many Spring Data modules including JPA—repositories. If you dive in and start using the template's rich set of operations, it can be challenging to learn the breadth of this API. It's kind of like a

typesafe way to write JPQL queries. If you've never written one before, you have much to learn.

Many operations are simple and almost universal. Saving, finding, and deleting entries doesn't require you to plumb the deepest recesses of JPA and H2. And the concepts are transferable should you choose a different data store on your next Spring Boot-based project.

Creating a repository isn't hard at all.

### Data repository for Item objects

```
import org.springframework.data.repository.CrudRepository;  
  
public interface ItemRepository extends CrudRepository<Item, Integer>  
{  
}  
}
```

This shows an `ItemRepository` extending Spring Data Commons' `CrudRepository` with the following characteristics:

- The first generic parameter `Item` describes the type that is stored and retrieved.
- The second generic parameter `String` describes the key.
- `ItemRepository` itself is an interface, meaning there is no code to write.

That's right. You have extended a Spring Data type interface, making your own interface. What did you inherit from this `CrudRepository`?

- `save()` and `saveAll()`
- `findById()`, `findAll()`, and `findAllById()`
- `existsById()`
- `count()`
- `deleteById()`, `delete()`, and `deleteAll()`

These methods provide a rich set of CRUD-based operations. And you don't have write a thing to use them! As for custom queries, you'll get to that further down in this chapter.

The declaration of your custom extension simply forces *what* the types of identifiers and domain objects are.

## Loading Test Data

So you need to load some test data. Great! What would do the job best? Why not...use the repository you just created?

If you grabbed a copy of your delightful `ItemRepository`, you'd write something like this.

Saving a single `Item` into the inventory collection, incorrectly

```
itemRepository.save(new Item("Alf alarm clock", 19.99))
```

And that's all you need!

Assuming you are all clear for using Spring Data to insert some data (what's better than Spring Data itself, right?), you could create a data-loading class.

Database loading component that uses a blocking API

```
@Component ①
public class RepositoryDatabaseLoader {

    @Bean ②
    CommandLineRunner initialize(BlockingItemRepository repository) {
        ③
        return args -> { ④
            repository.save(new Item("Alf alarm clock", 19.99));
            repository.save(new Item("Smurf TV tray", 24.99));
        };
    }
}
```

- ① `@Component` is a Spring Framework annotation that registers this class during Boot's component scanning phase.
- ② `@Bean` registers this `CommandLineRunner` as a bean. A `CommandLineRunner` is a Spring Boot component that executes after the application has started. It is guaranteed to be activated after all components are registered and activated.
- ③ The `initialize` method has `ItemRepository` as an input, telling Spring Framework to find a bean of this type and inject it.
- ④ The code inside the `initialize` method is another lambda function. It allows you to define a `CommandLineRunner` without having to whip up an anonymous subclass. The code then uses the `repository` to create some `Item` objects.

This code nicely creates two items, your top-selling Alf alarm clock and your cuddly Smurf TV tray, and to load them into H2.



CommandLineRunner objects are NOT guaranteed to start in any particular order. Do NOT attempt to coordinate between them.

## Defining a repository for Cart objects

```
public interface CartRepository extends CrudRepository<Cart, String>
{
}
```



Some of this lingua franca surrounding "does an Item make sense in the context of a Cart object" is captured in a tactic known as **Domain-Driven Design** or **DDD**. In fact, if you grab a copy of *Domain-Driven Design* by Eric Evans, you'll find a book that has stayed relevant for *years*. In fact, many of DDD's fundamental principles of data and repository design may ring truer today than when first explored sixteen years ago.

## Showing the cart

Before you start fiddling around with the cart, it would be useful to *show* the cart on the web page, ehh?

This begs you to retool that `HomeController` you slapped together in the previous chapter.

To do that, you first need to inject your new repositories.

`HomeController` getting repositories via constructor injection

```
@Controller ①
public class HomeController {

    private ItemRepository itemRepository;
    private CartRepository cartRepository;

    public HomeController(ItemRepository itemRepository, ②
        CartRepository cartRepository) {
        this.itemRepository = itemRepository;
        this.cartRepository = cartRepository;
    }

    ...
}
```

① `@Controller` signals this as a Spring Web controller that serves up templated views.

② The constructor is used by Spring to inject the `itemRepository` and the `cartRepository`. This is known as **constructor injection** and is a highly recommended tactic by the Spring team.

With this in place, you can post a more sophisticated listing of your inventory and the cart.

Overhauled rendering for a Cart at the root URL

```
@GetMapping  
String home(Model model) { ①  
    model.addAttribute("items", //  
        this.itemRepository.findAll()); ②  
    model.addAttribute("cart", //  
        this.cartRepository.findById("My Cart") ③  
            .orElseGet(() -> new Cart("My Cart")));  
    return "home";  
}
```

- ① This version returns `String`, the name of the template to render.
- ② `model.addAttribute(...)` is where you provide data to the template.
- ③ `.findById(...).defaultIfEmpty(...)` is a Spring Data Recipe™ where you retrieve the cart from H2, but if not found, fall back to creating a new `Cart`.

By supplying both `itemRepository.findAll()` (which provides a `Iterable<Item>`) and the `Cart` to a template engine like Thymeleaf, you only need add a smidgeon of HTML to `home.html`.



Notice how the name of the cart is `1`? Carts are usually associated with a user's session. You aren't there yet (but you will be later in this book!) For now, letting `1` be stand-in for the "session" of the cart is good enough.

First of all, add an HTML `<table>` to display the current inventory.

HTML `<table>` showing entire inventory of Item objects

```

<h2>Inventory Management</h2>
<table>
    <thead>
        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>Price</th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="item : ${items}">
            <td th:text="${item.id}"></td>
            <td th:text="${item.name}"></td>
            <td th:text="${item.price}"></td>
            <td>
                <form method="post" th:action="@{/add/' + ${item.id}}">
                    <input type="submit" value="Add to Cart" />
                </form>
            </td>
            <td>
                <form th:method="delete" th:action="@{/delete/' +
${item.id}}">
                    <input type="submit" value="Delete"/>
                </form>
            </td>
        </tr>
    </tbody>
</table>

```

This HTML `<table>` uses Thymeleaf's `th:each` iterator to create a separate row for each inventory Item.

There is an HTML `<form>` to POST new entries to the cart as well as one to DELETE things from the inventory.

HTML supports two actions: GET and POST. Other operations require a bit of a trick. Thymeleaf allows you to widen HTML forms by providing a `th:method="delete"` operation. When rendered as HTML, it turns it into a POST call with a hidden input, `<input type="hidden" name="_method" value="delete"/>`. Spring MVC has a special filter that hooks this web call onto a controller method with the `@DeleteMapping` annotation.

Finish your HTML template by adding another section to display the shopping cart.

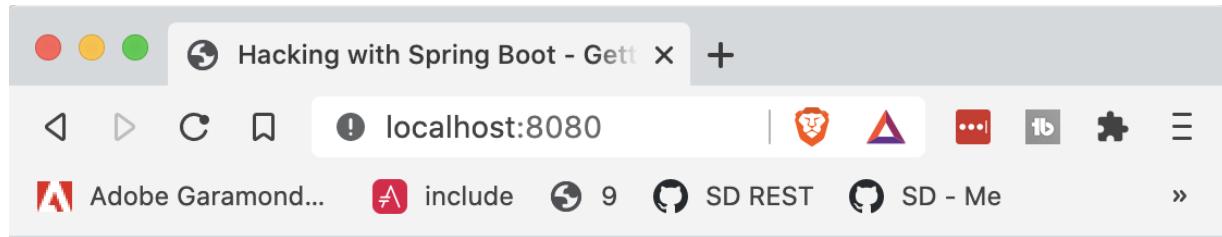
HTML `<table>` that displays the current shopping cart

```
<h2>My Cart</h2>
<table>
    <thead>
        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>Quantity</th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="cartItem : ${cart.cartItems}">
            <td th:text="${cartItem.item.id}"></td>
            <td th:text="${cartItem.item.name}"></td>
            <td th:text="${cartItem.quantity}"></td>
        </tr>
    </tbody>
</table>
```

This HTML `<table>` uses Thymeleaf's `th:each` iterator to cycle through all of the `Cart` object's `CartItem` entries, listing `id`, `name`,

and quantity. Notice how Thymeleaf navigates to properties via dot notation?

With this in place, you can now fire up the application inside your IDE (or with `./mvnw spring-boot:run`) and take a peek at `localhost:8080`.



# Welcome to Hacking with Spring Boot!

## Inventory Management

<b>Id</b>	<b>Name</b>	<b>Price</b>		
1	Alf alarm clock	19.99	<a href="#">Add to Cart</a>	<a href="#">Delete</a>
2	Smurf TV tray	24.99	<a href="#">Add to Cart</a>	<a href="#">Delete</a>

## My Cart

<b>Id</b>	<b>Name</b>	<b>Quantity</b>
-----------	-------------	-----------------

Right away, you can see the two pre-loaded inventory items. And at the bottom is a nice, empty cart.

## Adding items to a cart

With these building blocks, it's time to load up the cart. This is the bread and butter of any e-commerce system, so let's get it right.

What are you shooting for?

- You need to retrieve the current cart. If none exists, create one.
- If the item is already in the cart, increment its quantity. If not, add a new entry of 1.
- Save the updated cart.

To get this cart moving, you need to code that `add/{itemId}` operation.

Adding an Item to the Cart

```
@PostMapping("/add/{id}") ①
String addToCart(@PathVariable Integer id) { ②
    Cart cart = this.cartRepository.findById("My Cart") //
        .orElseGet(() -> new Cart("My Cart")); ③

    cart.getCartItems().stream() //
        .filter(cartItem -> cartItem.getItem().getId().equals(id)) //
        .findAny() ④
        .map(cartItem -> {
            cartItem.increment();
            return cart;
        }) //
        .orElseGet(() -> { ⑤
            Item item = this.itemRepository.findById(id)
                .orElseThrow(() -> new IllegalStateException("Can't
seem to find Item type " + id));
        });
    }
} //
```

```

        cart.getCartItems().add(new CartItem(item, cart));
        return cart;
    });

this.cartRepository.save(cart); ⑥

return "redirect:/"; ⑦
}

```

Wow! That's a lot of code. Let's dig through the key parts.

- ① `@PostMapping` maps this method onto `POST /add/{id}`. This lets the web page put a button on every item.
- ② `@PathVariable` extracts the `{id}` part of the route into the method parameter named `id`.
- ③ There's that Spring Data Recipe™ again! Find "My Cart" from H2 or fall back to a newly-created instance via `defaultIfEmpty()`.
- ④ After getting a cart, the first step is figuring out if the cart already has the item. The classical approach is to use a for-each loop, but thanks to the Stream API, you can filter over all the `CartItem` objects, and see if any of their `Item` objects match the incoming `id`. The Stream API gives you an `Optional<CartItem>` through the `findAny()` method. From there, increment the `CartItem` and then return the updated `Cart`.
- ⑤ If the `Optional` is actually empty, retrieve the `Item` from H2 and wrap it inside a `CartItem`, which defaults to a quantity of 1. Add it to the `Cart` and return the `Cart`. Because this only happens if not already in the cart, the entire operation is wrapped inside a lambda function.
- ⑥ Take the updated `Cart` and save it back to H2.
- ⑦ Wrap things up with a `redirect:/` which tells Spring MVC to issue an HTTP redirect signal back to `/`.

If you aren't used to working with Java's `Optional` type and it's Stream API, then what you just read could be a bit heady.

`Optional` is simply a wrapper around a value that may or may not be there. In essence, you either extract the value itself, or if it was `EMPTY`, you get an alternative value.

Java's Stream API gives you a new way to iterate over collections and look for what you need. There are, in fact, entire books on this API, so we won't cover it in depth. But by the end of this book, you should be familiar with the basics.

For comparison, the traditional way to see if your `Cart` had an `Item` with an `id` value of, say, 5, you'd write this.

#### Classic cart retrieval

```
boolean found = false;

for (CartItem cartItem : cart.getCartItems()) {
    if (cartItem.getItem().getId().equals("5")) {
        found = true;
    }
}

if (found) {
    // increment
} else {
    // add new CartItem
}
```

This is straight forward. You start with a boolean of false and then loop over all the `CartItem` objects in the `Cart`. Further down, you either increment or add, depending on whether it already exists.

So why doesn't stream-based programming do this? The side effect of imperative programming are all the local variables you use. You build up tiny bits of state and are constantly moving content in and out of other APIs.

The risk that something doesn't line up properly increases with every intermediate variable. As strong as Java's type system is, history has shown it's not strong enough.

Using Java's Stream API, this would be the upgraded way of "finding out."

### Streaming cart retrieval

```
if (cart.getCartItems().stream() //  
    .anyMatch(cartItem -> cartItem.getItem().getId().equals("5"))) {  
    // increment  
} else {  
    // add new CartItem  
}
```

If you are wholly unfamiliar with the Stream API, you may need to read this fragment of code a few times.

- `cart.getCartItems().stream()` kicks things off by giving you a handful of operations using this lazy iterator.

- `.anyMatch()` is a function that looks at each object of the stream, and yields a boolean. The first one that returns true breaks out of the stream and returns control to you.

Here's the big thing: No Intermediate State. No found flag to manage, and no risk of initializing it incorrectly and flipping it improperly. The logical outcome is tightly coupled to the matching function, not your ability to manage local variables. And that's a good thing!

By shifting to a paradigm where you chain together functions, with the output of one fed immediately into the input of the next, has a strange effect. While it may feel harder to write your flows at first, the strictness tends to me more "correct" by the time you finish.

## Wrapping things inside a service

Have you read through `addToCart()`? It's quite a bit of code. In fact, if you're still fuzzy, I suggest you read through it once more.

Why is that code so dense? Because you don't have all those temp variables, and intermediate bits of state. By cutting that out, what's left are operations that do real lifting. Each line moves things forward.

A consequence of this is that your web controllers can quickly get too heavy. Phil Webb, lead for Spring Boot, recommends you keep controllers light and focused on handling web requests, not business logic.

All that code involved to retrieve cart and add items should probably be pushed down into a service.

Adding to a cart after refactoring it into a @Service bean

```
@Service ①
class CartService {

    private final ItemRepository itemRepository;
    private final CartRepository cartRepository;

    CartService(ItemRepository itemRepository, CartRepository
    cartRepository) { ②
        this.itemRepository = itemRepository;
        this.cartRepository = cartRepository;
    }

    Cart addToCart(String cartId, Integer itemId) { ③
        Cart cart = this.cartRepository.findById(cartId) //
            .orElseGet(() -> new Cart(cartId));

        cart.getCartItems().stream() //
            .filter(cartItem ->
        cartItem.getItem().getId().equals(itemId)) //
            .findAny() //
            .map(cartItem -> {
                cartItem.increment();
                return cart;
            }) //
            .orElseGet(() -> {
```

```

        Item item = this.itemRepository.findById(itemId) //  

            .orElseThrow(() -> new IllegalStateException("Can't  

        seem to find Item type " + itemId));  

        cart.getCartItems().add(new CartItem(item, cart)); ④  

        return cart;  

    });

    return this.cartRepository.save(cart); ⑤
}
}

```

- ① `@Service` identifies this component as a service holding no state of its own. This class will be automatically picked up by Spring Boot's classpath scanning and made a full-fledged component.
- ② Just like in the controller, this service bean gets its repository definitions by **constructor injection**.
- ③ The incoming parameters include the `Cart id` and the `Item id`.
- ④ Having fetched the cart, and possibly thrown an exception, simply add it to the `Cart`.
- ⑤ With the cart built up, save it back to H2 using the repository.

By moving this core procedure of adding or incrementing the `Item` in a `Cart` into a service, you can now simplify the controller. Assuming you have already added `private final CartService cartService` and initialized it in the constructor, the `addToCart(...)` web handler can be rewritten.

`addToCart` rewritten to use `CartService`

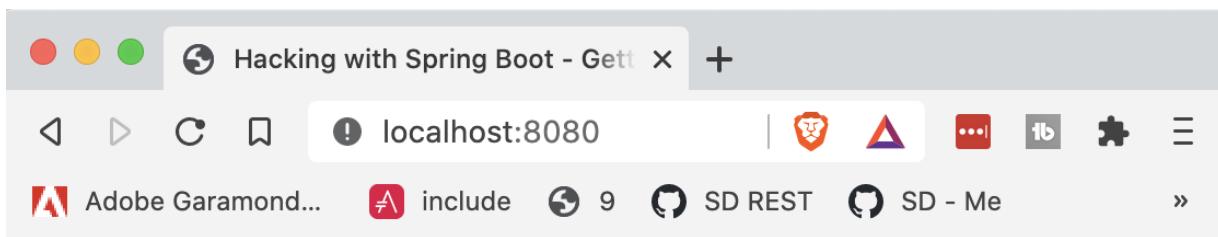
```

@PostMapping("/add/{id}")
String addToCart(@PathVariable Integer id) {
    this.cartService.addToCart("My Cart", id);
}

```

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

With this code in place, relaunch the application and visit <http://localhost:8080>. You'll see a listing of inventory as before. Now click on the Add links a few times. You should build up a cart similar to this one.



# Welcome to Hacking with Spring Boot!

## Inventory Management

<b>Id</b>	<b>Name</b>	<b>Price</b>		
1	Alf alarm clock	19.99	<a href="#">Add to Cart</a>	<a href="#">Delete</a>
2	Smurf TV tray	24.99	<a href="#">Add to Cart</a>	<a href="#">Delete</a>

## My Cart

<b>Id</b>	<b>Name</b>	<b>Quantity</b>
1	Alf alarm clock	4
2	Smurf TV tray	3

## Querying the database

So far, you've both stored and retrieved data from H2, but it hasn't been...what's the word...fantastic?

Looking up things by `id` isn't the real ticket. After all, in a real world e-commerce site, you can't list your entire inventory and wait for the customer to select the item that way.

No, you need customers to enter criteria into a search box. Using the input, query the `items` collection. In fact, let's do just that—write a custom query.



The following code examples can be found [here](#).

### Item repository with custom finder

```
public interface ItemRepository extends CrudRepository<Item, String>
{
    List<Item> findByNameContaining(String partialName);
}
```

This repository contains a custom finder that matches on the `Item.name` property. The `containing` clause makes it a partial match. And notice how you didn't have to write the code yourself?

Spring Data's secret sauce is that for 80% of the queries you'll need, you can use the tactic shown above—define a "finder"

and let the toolkit *derive the query* for you!

You don't have to speak JPQL or whatever QL is needed to talk to your data store. Spring Data will do it for you.

To paint a picture, let's make some tweaks. Imagine your Item object also had a description field.

Item object with description field

```
@Entity
public class Item {

    private @Id @GeneratedValue Integer id;
    private String name;
    private String description;
    private double price;

    // ...update getters, setters, equals,
    //      hashCode, and toString...
}
```

With this change, check out the query opportunities in the following table.

*Table 3. Spring Data query derivation options*

Finder Method	Description
<b>findByDescription(...)</b>	Query based on <b>description</b>
<b>findByNameAndDescription(...)</b>	Query based on <b>name</b> and <b>description</b>

Finder Method	Description
<b>findByNameAndDistributorRegion(...)</b>	Query based on the item's <b>name</b> and by a related distributor's <b>region</b>
<b>findTop10ByName(...)</b> or <b>findFirst10ByName(...)</b>	Query based on <b>name</b> , but only return the first ten entries
<b>findByNameIgnoreCase(...)</b>	Query by <b>name</b> , but ignore the case of the text
<b>findByNameAndDescriptionAllIgnoreCase(...)</b> )	Query by <b>name</b> and <b>description</b> , but ignoring the case of the text in ALL fields
<b>findByNameOrderByDescriptionAsc(...)</b>	Query by <b>name</b> , but order the results based on <b>description</b> in ascending order (or use <b>Desc</b> for descending order)
<b>findByReleaseDateAfter(Date date)</b>	Query based on <b>releaseDate</b> being after the <b>date</b>
<b>findByAvailableUnitsGreaterThan(int units)</b>	Query based on <b>availableUnits</b> being greater than <b>units</b>

Finder Method	Description
<code>findByAvailableUnitsGreaterThanOrEqualTo(int units)</code>	Query based on <b>availableUnits</b> being greater than or equal to <b>units</b>
<code>findByReleaseDateBefore(Date date)</code>	Query based on <b>releaseDate</b> being before the <b>date</b>
<code>findByAvailableUnitsLessThan(int units)</code>	Query based on <b>availableUnits</b> being less than <b>units</b>
<code>findByAvailableUnitsLessThanOrEqualTo(int units)</code>	Query based on <b>availableUnits</b> being less than or equal to <b>units</b>
<code>findByAvailableUnitsBetween(int from, int to)</code>	Query based on <b>availableUnits</b> being between <b>from</b> and <b>to</b>
<code>findByAvailableUnitsIn(Collection unitss)</code>	Query based on <b>availableUnits</b> being found in the supplied collection
<code>findByAvailableUnitsNotIn(Collection unitss)</code>	Query based on <b>availableUnits</b> NOT being found in the supplied collection

Finder Method	Description
<code>findByNameNotNull()</code> or <code>findByNameIsNotNull()</code>	Query based on <b>name</b> not being null
<code>findByNameNull()</code> or <code>findByNameIsNull()</code>	Query based on <b>name</b> being null
<code>findByNameLike(String f)</code> or <code>findByNameStartingWith(String f)</code> or <code>findByNameEndingWith(String f)</code>	Query based on input being a regular expression
<code>findByNameNotLike(String f)</code> or <code>findByNameIsNotLike(String f)</code>	Query based on input being a regex, with a JPA not applied.
<code>findByNameContaining(String f)</code>	For a string input, query just like <b>Like</b> . For a collection, query testing membership in the collection
<code>findByNameNotContaining(String f)</code>	For a string input, query like like <b>NotLike</b> . For a collection, query testing lack of membership in the collection
<code>findByNameRegex(String pattern)</code>	Query using <b>pattern</b> as a regular expression
<code>findByActiveIsTrue()</code>	Query by <b>active</b> being true

Finder Method	Description
<code>findByActiveIsFalse()</code>	Query by <b>active</b> being false
<code>findByLocationExists(boolean e)</code>	Query by <b>location</b> having the same boolean value as the input

Take your time to digest this table. It's a plethora of query opportunity. And as a bonus, all of these keywords can *also* be used crafting `deleteBy` methods!



Many of these operators also work with other supported data stores including MongoDB, Apache Cassandra, and Apache Geode. Be sure to check the specific reference guide.

While the previous table shows the keywords supported for JPA repository queries, the following list shows the various supported return types.

- Item (or Java primitive types)
- Iterable<Item>
- Iterator<Item>
- Collection<Item>
- List<Item>

- `Optional<Item>` (Java 8 or Guava)
- `Option<Item>` (Scala or Vavr)
- `Stream<Item>`
- `Future<Item>`
- `CompletableFuture<Item>`
- `ListenableFuture<Item>`
- `@Async Future<Item>`
- `@Async CompletableFuture<Item>`
- `@Async ListenableFuture<Item>`
- `Slice<Item>`
- `Page<Item>`
- `Mono<Item>` (Reactor-based)
- `Flux<Item>` (Reactor-based)

These return types combined with all the keywords offers a rich ability to write finders *without* writing the code.

Code you don't write has no bugs.

~ Greg L. Turnquist



Spring Data blocking APIs support `void` return types as well. Just in case you indulge in Reactor-based programming in the future, the equivalent is `Mono<Void>`, because the caller needs the ability to invoke `subscribe()`.

## When query derivation isn't enough

"What if I need a custom query?"

This question *always* comes up when I give a talk on Spring Data and demonstrate the speed I can write queries. Having developed a system at a previous job that had over 200 SQL queries, I understand the cost to fashion and maintain such data-centric operations.

While finder methods and query derivations let you leap ahead, fast, you will hit a wall. Inevitably, you must write some query that exceeds the limits of all those keywords.

Someone will sail in from the business office and request joining multiple collections/tables with complex criteria, with little more said than, "when can you have this ready?" (veiled wording for "we needed it yesterday").

This code shows how to write custom queries.  
Item repository with hand-written query

```
@Query("select i from Item i where i.name = ?1 and i.price = ?2")
List<Item> findItemsForCustomerMonthlyReport(String name, int price);

@Query("select i from Item i order by i.price")
List<Item> findSortedStuffForWeeklyReport();
```

Spring Data's `@Query` annotation overrides query derivation and applies your custom query. The return type (`List<Item>` in this case) is applied during the conversion process.



The name of the method doesn't matter when using `@Query`, so pick something that makes sense to you.

## Query by Example

Query derivation and hand-written queries can get you pretty far. But what happens when you add a filter option to your site? You want to let customers search on various fields that they get to choose?

Imagine you added the ability to search based on `name`. Not too hard. You create `findByName(String name)` in your repository and call it a day.

But management comes back. They want it to do partial matches. Again, not hard. Tweak that last query to become `findByNameContaining(String partialName)`.

You get a big thumbs up. And five minutes later, your success rewards you with another request.

They want you to let customers search on *name and description*. Gritting your teeth, you crank out `findByNameAndDescription(String name, String description)`. But then your boss reminds you to make everything partial and to match any case. So, your simple finder morphs into `findByNameContainingAndDescriptionContainingAllIgnoreCase(String partialName, String partialDescription)`.

Not only is that hard to type into a book...err...your IDE, it's complicated by the next requirement. UX testing shows that people want a radio button to toggle between **and** and **or**.

By now, your repository is looking like this.

Series of simple finders used to implement management's complex requests

```
// search by name
List<Item> findByNameContainingIgnoreCase(String partialName);

// search by description
List<Item> findByDescriptionContainingIgnoreCase(String partialName);

// search by name AND description
List<Item>
findByNameContainingAndDescriptionContainingAllIgnoreCase(String
partialName, String partialDesc);

// search by name OR description
List<Item>
```

```
    findByNameContainingOrDescriptionContainingAllIgnoreCase(String  
partialName, String partialDesc);
```

And to make use of all these finders, you have built a search service.

Leveraging your finders to build that complex filter

```
Iterable<Item> search(String partialName, String partialDescription,  
boolean useAnd) {  
    if (partialName != null) {  
        if (partialDescription != null) {  
            if (useAnd) {  
                return repository //  
  
.findByNameContainingAndDescriptionContainingAllIgnoreCase( //  
                    partialName, partialDescription);  
            } else {  
                return  
repository.findByNameContainingOrDescriptionContainingAllIgnoreCase(  
//  
                    partialName, partialDescription);  
            }  
        } else {  
            return repository.findByNameContaining(partialName);  
        }  
    } else {  
        if (partialDescription != null) {  
            return  
repository.findByDescriptionContainingIgnoreCase(partialDescription);  
        } else {  
            return repository.findAll();  
        }  
    }  
}
```

This switch nightmare navigates to the proper finder method based on which inputs are null and whether or not someone has selected and or or.

This thing is...horrendous! Maintenance will only get worse as more and more requests come in. And when your manager asks that you add support for three more fields, you go home contemplating moving to another company.

There's got to be a better way!

**Query by Example** is here to save the day!

With Query by Example, you dynamically assemble a **probe** and feed it to Spring Data. It renders the query you need. And no need to maintain gobs of finder methods.

To get started, you first need a repository that extends `QueryByExampleExecutor<T>`.



Thanks to Java's support for multiple interface inheritance, you can apply this to your existing repository definition.

For demonstration purposes, let's create a new repository definition.

`ItemByExampleRepository` using `QueryByExampleExecutor<T>`

```
public interface ItemByExampleRepository extends CrudRepository<Item, Long>, QueryByExampleExecutor<Item> {  
}  
}
```

Look ma, no finders!

If you navigate to Spring Data's `QueryByExampleExecutor`, you may find it quite illuminating.

Spring Data Common's query-by-example base

```
public interface QueryByExampleExecutor<T> {  
    <S extends T> Optional<S> findOne(Example<S> var1);  
  
    <S extends T> Iterable<S> findAll(Example<S> var1);  
  
    <S extends T> Iterable<S> findAll(Example<S> var1, Sort var2);  
  
    <S extends T> Page<S> findAll(Example<S> var1, Pageable var2);  
  
    <S extends T> long count(Example<S> var1);  
  
    <S extends T> boolean exists(Example<S> var1);  
}
```

Search for one or multiple T objects, with the option to sort. Also count and test for existence.

Each of these methods is focused on either finding one "thing" (an `Optional`) or many "things" (an `Iterable` or a `Page`). Each operation accepts an `Example`, i.e. a **probe**. And there are options to include a `Sort` or `Pageable` directive.

In certain scenarios you don't need the data itself, just the *number* of "things". In other use cases, you merely need to know whether or not any such data exists matching the probe's criteria.

Does this API appear a tad lightweight? Perhaps insufficient for your complicated needs? Standby, the power here is subtle.

The best way to illustrate this sophisticated feature is using the same example used to craft that horrendous `search()` function earlier. To recap management's crazy requirements, you were asked to support customer's filtering on name and description, with partial, case-ignoring matches, using either `and` or `or` based on a toggle.

Implementing it with Query by Example is shown below.  
Query by Example used to implement a customer-driven filter

```
Iterable<Item> searchByExample(String name, String description,
boolean useAnd) {
    Item item = new Item(name, description, 0.0); ①

    ExampleMatcher matcher = (useAnd ②
        ? ExampleMatcher.matchingAll() //
        : ExampleMatcher.matchingAny()) //
        .withStringMatcher(StringMatcher.CONTAINING) ③
        .withIgnoreCase() ④
        .withIgnorePaths("price"); ⑤

    Example<Item> probe = Example.of(item, matcher); ⑥
```

```
    return exampleRepository.findAll(probe); ⑦  
}
```

- ① Take the customer's arguments (either can be `null`) and populate an `Item`.  
Since `price` can't be `null`, populate it with `0.0`.
- ② Using Java's ternary operator, either match on *all* fields or *any* fields,  
based on their selection of `and` or `or`.
- ③ Use `StringMatcher.CONTAINING` to match on partials. (HINT: Check out  
`StringMatcher` for other cool options!)
- ④ Ignore case as well.
- ⑤ By default, `ExampleMatcher` ignores `null` fields. That doesn't work for a  
`double`, so explicitly ignore the `price` field.
- ⑥ Wrap the object using `Example.of(...)`, including the `matcher` to create your  
**probe**.
- ⑦ Exercise the query.

Query by Example API is not looking so "lightweight" now, eh?  
Not only could you express your entire needs with that probe.  
You can probably see how to alter things as more needs arrive.



It's left as an exercise to code a global search, i.e. the user enters  
their search in one text box, and the backend searches ALL fields  
(partial, case-ignoring) using Query by Example.

The feature is not complete until you hook it up to the web.  
Thanks to the power of Spring MVC, this is a piece of cake.  
Item repository with hand-written query

```

@GetMapping("/search") ①
String search( //
    @RequestParam(required = false) String name, ②
    @RequestParam(required = false) String description, //
    @RequestParam boolean useAnd, //
    Model model) {
    model.addAttribute("results",
    inventoryService.searchByExample(name, description, useAnd)); ③
    return "home"; ④
}

```

- ① This searching function responds to GET /search requests.
- ② It accepts several HTML query parameters (e.g. ?  
name=foo&description=bar). The first two parameters, if not specified, will render as null. The useAnd parameter is required and will cause an error if not supplied.
- ③ Invoke the inventory service's `findVariousItems` that you just defined, and use it to lazily populate the template's data model.
- ④ Using the `Model` container, declare the template view to render.

## Trade offs

In digging through Spring Data JPA, you have seen:

- Standard CRUD methods (`findAll` and `findById`)
- Custom finder methods (`findByNameContaining`)
- Query by Example
- `HibernateOperations`
- `@Query`-annotated finders

There are, in fact, even more options. Since this is *Hacking with Spring Boot* (and not *Hacking with Spring Data JPA*), this should be more than enough build a fully-armed and operational application.

It's important to realize that each of these tactics has their pros and cons. Let's review them.

Query Method	Pros	Cons
CRUD methods	<ul style="list-style-type: none"><li>▪ Already defined AND written</li><li>▪ Supports many return types</li><li>▪ Portable between data stores</li></ul>	<ul style="list-style-type: none"><li>▪ Limited to finding one or all</li><li>▪ Requires a custom interface for each domain object</li></ul>
Custom finders	<ul style="list-style-type: none"><li>▪ Intuitive</li><li>▪ Query is automatically written</li><li>▪ Supports many return types</li><li>▪ Portable between data stores</li></ul>	<ul style="list-style-type: none"><li>▪ Requires a custom interface for each domain object</li><li>▪ Complex queries can get unwieldy</li></ul>

Query Method	Pros	Cons
Query by Example	<ul style="list-style-type: none"> <li>▪ Query is automatically written</li> <li>▪ Useful when you don't know all the criteria up front</li> <li>▪ Also useable with MongoDB and Redis</li> </ul>	<ul style="list-style-type: none"> <li>▪ Requires a custom interface for each domain object</li> </ul>
HibernateOperations	<ul style="list-style-type: none"> <li>▪ Get maximum usage out of your data store</li> <li>▪ Does NOT require a custom interface</li> </ul>	<ul style="list-style-type: none"> <li>▪ Tightly couples you to the data store</li> </ul>
@Query-based finders	<ul style="list-style-type: none"> <li>▪ Lets you write pure JPQL</li> <li>▪ Avoids unwieldy method names</li> <li>▪ Available to all data stores</li> </ul>	<ul style="list-style-type: none"> <li>▪ Tightly couples you to the data store</li> </ul>

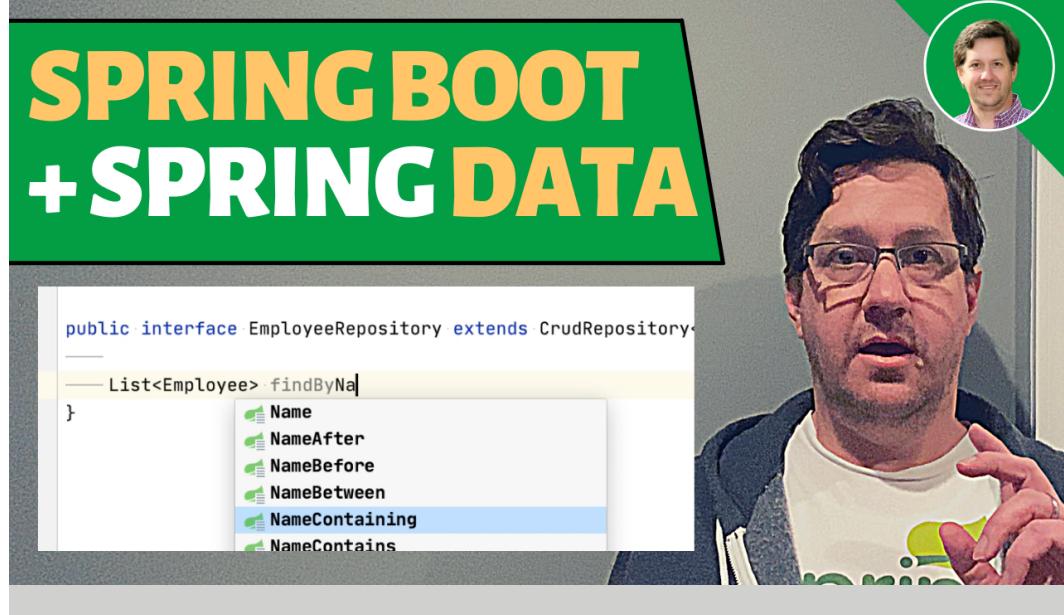
Essentially, you have to decide when you need that decoupling from the data store vs. the ability to hook in a highly-optimized query.



While these various tactics offer different levels of decoupling from the data store, migration isn't a heavily-exercised tactic. You should do a proper analysis when deciding what data store to use, because they all have various trade-offs. Things are more complex than simply "rewriting a few queries." In fact, the data store you choose can heavily influence how you define your domain objects. Evaluate each approach provided by Spring Data in terms of what best serves your application and domains instead of what offers the best "escape hatch."



Don't forget to check out my YouTube video [\*\*SPRING BOOT + SPRING DATA\*\*](#).



## Summary

In this chapter you:

- Defined your e-commerce application's domain objects.
- Created a repository to store and retrieve objects.
- Explored different ways to fashion custom queries.
- Wrapped the whole thing in a service to decouple it from the web layer.

In the next chapter, *Developer Tools for Spring Boot*, you will discover the various tools at your disposal to enhance the developer experience and learn how to get all you can out of your Spring Boot application!

3

---

# DEVELOPER TOOLS FOR SPRING BOOT

---

Spring Boot FTW is a beast ;)

~ Edward Beckett @edwardjbeckett

In the previous chapter, you learned how to use Spring Data. With Spring Boot wiring it all up for you, it was a snap to focus on defining domain objects and writing queries.

In this chapter, you are going to:

- Learn how to automatically restart your app as you make changes
- Utilize LiveReload to refresh the browser
- Use the tools provided by Spring Boot to help debug applications

Let's get to it!

# Starting your application…faster

A big thing developers have complained about in the past is having to constantly stop and start their application as they make changes. This used to take for-ever when you had to build a WAR file and deploy it to an application server. IDEs did their best by automatically carrying out this process, but it was visible nonetheless.

Spring Framework, long before Boot entered the scene, made the first foray into this problem by switching from full-blown app servers to servlet containers. This made the entire development process lighter and more nimble.

But alas, it wasn't enough. Or shall we say, once people got used to that, they started looking at other ways to speed up their cycles.

Enter Spring Boot.

When Spring Boot debuted back in 2014, it introduced a landmark change—embedded servlet containers. Instead of building a WAR file and deploying it to an already installed servlet container like Apache Tomcat, it brought the container *with it*.

By putting the container *inside* the application, Spring Boot not only sped up the process of launching an application, it inverted the whole concept of application deployment.

From that time forward, applications were no longer beholden to whatever servlet container their Ops people had installed. Instead, Spring Boot could specify *what* container was needed, and *how* it should run. This made it possible to ship a plain old JAR file, and all that was needed was a JVM somewhere on the target machine.

While not a huge increase in startup speed, it was a discernable improvement.

And people were impressed. In fact, renowned conference speaker [Eberhard Wolff](#) even gave a talk (shared by [yours truly](#)) titled "Java App Servers are Dead."

Yet, even with this vast improvement in the developer experience, the Spring Boot team continued seeking better, faster ways to re-start applications, over and over.

## Say "hi" to Developer Tools

Not resting on their laurels, the Spring Boot team introduced a new module several versions back called "Developer Tools", or simply "DevTools."

This module provides multiple features:

- Automatic restart and reload of your application
- Property defaults

- Logging of changes in autoconfiguration
- Exclusion of static resources
- LiveReload support

In order to add DevTools to your project, apply the following changes to your build file.

#### Adding Spring Boot DevTools to the build

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

In Maven, you set it as an optional dependency, so it doesn't end up in your production code. This is also recommended so that `spring-boot-devtools` isn't transitively applied to other code beyond your own. You also use the `runtime` scope since it isn't required to compile your code.

`spring-boot-devtools` looks at how your application is launched. If it spies things running via `java -jar` or from a special classloader (such as a cloud provider), then the module will deem it a "production" application and NOT use the DevTools features. But run inside your IDE or using Maven's `spring-boot:run` will signal it to run in "development" mode with all the features enabled (which you'll explore further in this chapter).



[Check the reference docs](#), and you'll find the incantation to use `spring-boot-devtools` with Gradle.

With this change applied, let's dig through this stack of developer goodies!

## Automatic restarts and reloads

To continue the story of speeding up application development, the Spring Boot team added the ability to detect changes in user code and *restart* your application.



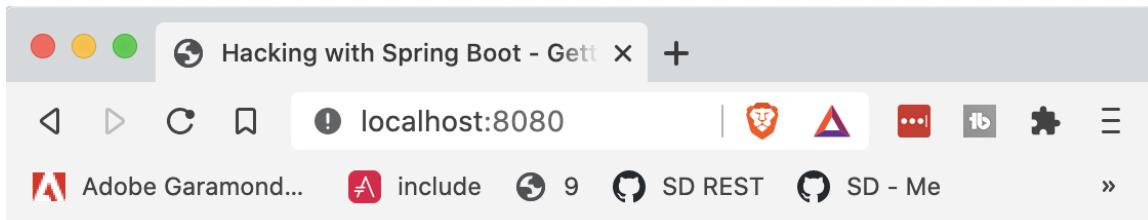
**Restart vs. Reload** - DevTools loads your code into one classloader and 3rd-party libraries into a separate classloader. When the application is *restarted*, the classloader with your code is thrown away and a new one created. But the 3rd-party classloader is retained. This makes it possible for cycles to be much faster than a "cold" start.

However, Spring Boot has limits on how far it can go. For maximum *reload* capability, you may have to investigate Java agent solutions like JRebel.

If you run the code you built in *Chapter 2, Data Access with Spring Boot* with Spring Boot's DevTools module added as

shown above, you can make changes to the code and see them take effect.

1. Launch the application.
2. Visit <http://localhost:8080>



# Welcome to Hacking with Spring Boot!

## Inventory Management

<b>Id</b>	<b>Name</b>	<b>Price</b>		
1	Alf alarm clock	19.99	<button>Add to Cart</button>	<button>Delete</button>
2	Smurf TV tray	24.99	<button>Add to Cart</button>	<button>Delete</button>

## My Cart

<b>Id</b>	<b>Name</b>	<b>Quantity</b>

3. If you'll remember, the `HomeController.home()` method from *Chapter 2* looked like this.

```
@GetMapping  
String home(Model model) { ①  
    model.addAttribute("items", //  
        this.itemRepository.findAll()); ③  
    model.addAttribute("cart", //
```

```
    this.cartRepository.findById("My Cart") ④
        .orElseGet(() -> new Cart("My Cart")));

    this.itemRepository //  

        .findAll() //  

        .forEach(System.out::println);
    return "home";
}
```

Copy the `itemRepository.findAll()`, with an `.forEach(System.out::println)` appended that prints each `Item` object:

```
this.itemRepository
    .findAll()
    .forEach(System.out::println)
```

4. Execute a **Save** command (or **Build Project** command) to force the IDE to signal Spring Boot to perform a restart.
5. Monitor the console output to see the app restart.
6. Refresh the browser page and verify that the fetched `Item` objects are logged to the console.



In my version, the main application took about 2.8 seconds to launch, while the application restart took less than a second. While not a big improvement, the difference grows as you add more components and build a "real" application.

As mentioned in the steps, the actual signal may vary from IDE to IDE. Some IDEs support "Save" or "Save All" commands. Others require you to force a "Rebuild."

Once Spring Boot DevTools gets the signal, it will attempt to restart. If the changes aren't too huge, it should go smooth.

## Exclusion of static resources

By default, Spring Boot ignores changes to:

- /META-INF/maven
- /META-INF/resources
- /resources
- /static
- /public
- /templates.

That's because most underlying web technologies can handle changes without requiring a server restart.

Set a property if you need to revise the paths that are ignored for a server restart.

Tweaking the files that trigger a restart

```
spring.devtools.restart.exclude=static/**,public/**
```

This setting will *only* ignore changes underneath /static and /public. Changes anywhere else will cause a server restart.



This is especially useful if you're building a frontend and you need to resolve a conflict between your toolkit (NodeJS, Webpack, etc.) and DevTools.

It's also possible to disable server restarts by setting `spring.devtools.restart.enabled` to `false`.

## Disabling caches in debug mode

Many components integrated through Spring Boot have various means of caching. For example, some template engines cache compiled templates. This is handy in production, but in development it can get in your way.

Spring Boot provides a property for Thymeleaf, `spring.thymeleaf.cache`, that you can set to `false` if you want to disable it. There are other properties you may wish to switch as well.

That's nice. But switching this property back and forth can get cumbersome, fast.

That's why when you add `spring-boot-devtools` to your build file, and you run things in development mode (from your IDE or

using Maven), it will flip ALL of these properties *automatically*. This feature is called "property defaults."

In fact, when you run your application, you can see this displayed on the console.

Evidence of property defaults being enabled by DevTools

```
2021-01-19 22:07:41.267 : Devtools property defaults active! Set  
'spring.devtools.add-properties' to 'false' to disable
```

This alerts you to "property defaults" being activated, and also gives you the hint on how to disable it if you wish.

At any time if you wish to see all the properties that Boot adjusts, just open `DevToolsPropertyDefaultsPostProcessor` inside your IDE.

## Logging extra web activity

For both Spring MVC and Spring WebFlux web applications, you can activate a bonus logging group, `web`, in your `application.properties` file.

Just add the following.

Increase web-level logging with developer tools

```
logging.level.web=DEBUG
```

Instead of having to dig down and uncover some package or class-level debugging setting, this simple logger can be activated

to give you a glimpse of what's happening from a web perspective.

For example, look what happens when you add a new item to the cart.

### Enhanced logging for a web request

```
2020-01-19 : HTTP POST "/add/5e252896a31dc10cddc35afd"
2020-01-19 : Mapped to
com.greglturnquist.hackingspringboot.classic.HomeController#addToCart
(String)

...H2 interactions trimmed for brevity...

2020-01-19 : View name 'redirect:/', model {}
2020-01-19 : Completed 303 SEE_OTHER
2020-01-19 : HTTP GET "/"
2020-01-19 : Mapped to
com.greglturnquist.hackingspringboot.classic.HomeController#home()
2020-01-19 : Using 'text/html' given [text/html,
application/xhtml+xml, image/webp, image/apng, application/xml;q=0.9,
application/signed-exchange;v=b3;q=0.9, */*;q=0.8] and supported
[text/html, application/xhtml+xml, application/xml, text/xml,
application/rss+xml, application/atom+xml, application/javascript,
application/ecmascript, text/javascript, text/ecmascript,
application/json, text/css, text/plain, text/event-stream]
2020-01-19 : Completed 200 OK
```

With this extra bit of web detail, it's easy to see what HTTP interactions happened, what controller methods were exercised, and the final status of everything.



Parts of the logging output, like the time, logging level, and thread id, were removed for readability.

## Logging of changes in autoconfiguration

One of Spring Boot's killer features is **autoconfiguration**. Based on the existence of various beans, property settings, and libraries on the classpath, Spring Boot will configure things. Sensible defaults usually benefit you, but sometimes things go astray.

If you're trying to divine what is happening, it can be handy to get Spring Boot to tell you what it's done. While you can ask for the report showing *all* the decisions Spring Boot has made, the volume of information can be overwhelming.

It's why in Spring Boot 2, the team started tracking *changes* in autoconfiguration. If you add a bean and it causes the autoconfiguration settings to add or remove a bean, it will show you that autoconfiguration delta.

## LiveReload support

Another built-in feature of Spring Boot DevTools is an embedded [LiveReload server](#). LiveReload performs a simple task—reload the web page when the server restarts. Many web

stacks support LiveReload. And with Spring Boot, you can do the same.

That's right, this feature clicks the reload button on your browser. As simplistic as that may sound, it really speeds up your cycles when working out the kinks in templates and web handlers.

To do this, the backend needs to run a server and your browser needs the [LiveReload plugin](#).



You can only run one LiveReload server on your machine at a time. Make sure you're not running some other web app that has LiveReload enabled before powering up your Spring Boot application.

## Summary

In this chapter you:

- Learned how to add Spring Boot Developer Tools to your project.
- Tinkered with automatically restarting the application, while making live changes to the code.
- Saw it flip certain properties to STOP caching when used in developer mode.

- Took a peek at Boot's built-in support for LiveReload.

In the next chapter, *Testing with Spring Boot*, you will discover how testing is a first-class citizen for Spring Boot. You'll use various techniques to test at various levels.

4

---

# TESTING WITH SPRING BOOT

---

More testing, more testing in Spring Boot  
;)

~ Alexander Orlovsky @ProJavaOrlovsky

In the previous chapter, you learned how to leverage Spring Boot's Developer Tools to speed up development cycles.

In this chapter, you'll dig in further and learn how to test applications. Maybe that sounds strange. After all, is testing special? No and yes.

"No" in the sense that whether or not you are coding with Spring Boot, tools like JUnit, AssertJ, and others are critical to writing solid apps.

But "yes" in the sense that Spring Boot and the whole Spring team bring additional teams giving you more options to test at various levels working *with* the power of Spring Boot.

In this chapter, you are going to:

- Discover Spring Boot's ability to run embedded container tests.
- Use **slice testing** to code tests that are somewhere between unit and integration.

Let's do this.

## Writing unit tests

The simplest, fastest, and easiest way to test anything is at the unit level. A "unit" is most commonly expressed in Java as a single class. In essence, verifying a single class acts properly by stubbing out any external collaborators defines a unit test.

Why is this the simplest way to test?

Because there is little machinery needed to put this together. In fact, you can do this with *no* machinery. However, it's handy to use something like JUnit 5 to at least to gather the results. In addition to that, there are powerful assertion libraries to help verify results.

---



### JUnit 4 vs. JUnit 5. Which one do I need?

That's easy. As rock-solid as JUnit 4 is, it's effectively feature-complete. All new development by the JUnit team is being poured into JUnit 5. When starting anything new, I'd pick JUnit 5.

If you have an existing system with *lots* of test cases written using JUnit 4, the transition can cost you a bit. However, it's not that hard to migrate. Last year, I migrated Spring HATEOAS and it only took one day to convert over 600 test cases.

What does Spring Boot bring to the table to assist, if no assistance is required?

Spring Boot upholds the importance of testing so much that a suite of test libraries comes pre-configured. You merely have to add `spring-boot-starter-test` to your build file and the following test libraries are automatically added:

- Spring Boot Test
- JsonPath
- JUnit 5
- AssertJ
- Mockito
- JSONassert
- Spring Test



When you create a project using the Spring Initializr ([start.spring.io](https://start.spring.io)), `spring-boot-starter-test` is added by *default*. You don't even have to think about it. Testing is set up and ready to go.

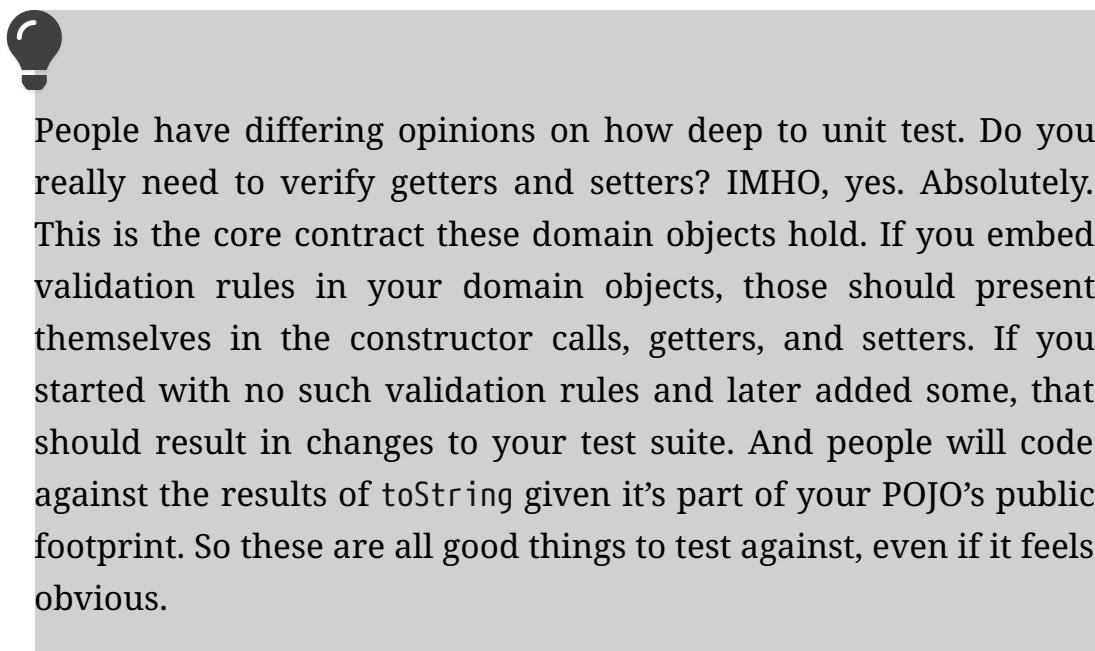
The simplest type of unit test is where there are few interactions you must capture. For example, the domain objects are probably the easiest given this is the basis for your whole application. Whether you are using an anemic approach with no business logic but possibly validation checks, or something richer, this layer should have no dependencies. Testing at this level is easiest given there should be no collaborators.

### Testing domain objects

```
class ItemUnitTest {  
  
    @Test ①  
    void itemBasicsShouldWork() {  
        Item sampleItem = new Item(1, "TV tray", "Alf TV tray", 19.99);  
        ②  
  
        // Test various aspects using AssertJ ③  
        assertThat(sampleItem.getId()).isEqualTo(1);  
        assertThat(sampleItem.getName()).isEqualTo("TV tray");  
        assertThat(sampleItem.getDescription()).isEqualTo("Alf TV tray");  
        assertThat(sampleItem.getPrice()).isEqualTo(19.99);  
  
        assertThat(sampleItem.toString()).isEqualTo( //  
            "Item{id='1', name='TV tray', description='Alf TV tray',  
            price=19.99}");  
  
        Item sampleItem2 = new Item(1, "TV tray", "Alf TV tray", 19.99);
```

```
        assertThat(sampleItem).isEqualTo(sampleItem2);
    }
}
```

- ① @Test tells JUnit this is a test method.
- ② Create an instance of the Item object.
- ③ Use your favorite assertion library (AssertJ in this case) to verify the domain object works as expected.



Run it inside your favorite IDE, and checkout the results.

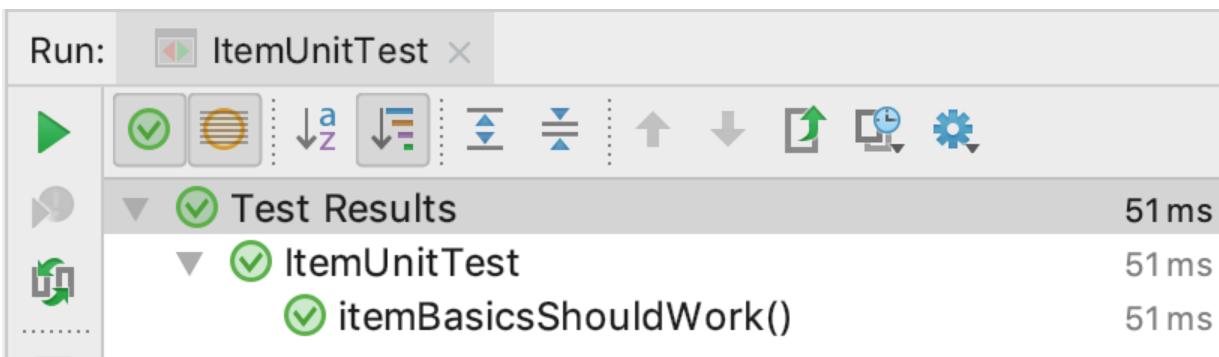


Figure 1. Running your tests inside the IDE

Domain objects aren't too hard to test. Things get trickier when exercising code that interacts with other components. A good example is your `InventoryService`. That class contains several operations that have business logic, while also interacting with external collections through repositories.

On top of that, the service is the first level that has collaborators. How, exactly, do you exercise *that*?

Never fear, Spring Boot + various test technologies will provide you with the tools to empower JUnit to verify all is well.

To sharpen your focus, let's hone in on `InventoryService.addItemToCart(...)` and test that one. You can start by creating a new test class called `InventoryServiceUnitTest` like this:

#### Creating a test class

```
@ExtendWith(SpringExtension.class) ①
class InventoryServiceUnitTest { ②
    ...
}
```

① `@ExtendWith` is JUnit 5's API to register custom test handlers.

`SpringExtension` is Spring Framework's hook for Spring-specific test features.

② By combining the name of the class under test (`InventoryService`) with the scope of the test (`UnitTest`), the name clearly denotes *what* is being tested in this class.

With your test class declared, it's now important to identify *what* is being tested...and what is not. This is known as the *class under test*. And for unit testing, any outside service or component is dubbed a *collaborator* and should be mocked or stubbed away.

So inside `InventoryServiceUnitTest`, declare these mock collaborators.

Distinguishing the class under test from its collaborators

```
InventoryService inventoryService; ①  
@MockBean private ItemRepository itemRepository; ②  
@MockBean private CartRepository cartRepository; ②
```

- ① `InventoryService` is the *class under test*. It has no special annotations and will be initialized further down.
- ② `ItemRepository` and `CartRepository` are external components that get injected into `InventoryService`, making them collaborators. They are both annotated with `@MockBean`, a Spring Boot test annotation that signals them to be wrapped as Mockito-based mocks.

There are entire books written about Mockito as well as test mocks. Since this is *Hacking with Spring Boot 2.4: Classic Edition*, the focus isn't on all the details of testing, but instead how Spring Boot speeds up your ability to write tests.

And in this case, Spring Boot provides the `@MockBean` annotation. It has two key functions.

- `@MockBean` saves you from having to write this.

## Manually crafting a mock

```
@BeforeEach  
void setUp() {  
    itemRepository = mock(ItemRepository.class);  
    cartRepository = mock(CartRepository.class);  
}
```

You could say you're swapping one line of code for another. But in truth, `@MockBean` is more concise and leverages existing knowledge of the collaborators' types.

- On a more subtle level, `@MockBean` clearly denotes the collaborators. It's semantically easier to grok what is being tested and what is not.

With a clear declaration of the *class under test* vs. the *collaborators*, it's time to configure things. Shown below is one way to pre-configure these test fields *before* running any actual tests.

## Setting things up

```
@BeforeEach ①  
void setUp() {  
    // Define test data ②  
    Item sampleItem = new Item(1, "TV tray", "Alf TV tray", 19.99);  
    CartItem sampleCartItem = new CartItem(sampleItem, null);  
    Cart sampleCart = new Cart("My Cart",  
        Collections.singletonList(sampleCartItem));  
    sampleCartItem.setCart(sampleCart);
```

```
// Define mock interactions provided  
// by your collaborators ③  
  
when(cartRepository.findById(anyString())).thenReturn(Optional.empty());  
  
when(itemRepository.findById(anyInt())).thenReturn(Optional.of(sampleItem));  
    when(cartRepository.save(any(Cart.class))).thenReturn(sampleCart);  
  
    inventoryService = new InventoryService(itemRepository,  
cartRepository); ④  
}
```

- ① `@BeforeEach` is JUnit 5's annotation that runs this `setUp()` method before each individual test case in this class.
- ② As a first step, define your test data. In this case, a `Cart` with a single `Item`.
- ③ With test data in hand, you can now define the mock interactions using Mockito.
- ④ Finally, create the *class under test*, constructing it with the mock *collaborators*.



If you need a method to run before ALL test methods in a given test class, use `@BeforeAll`.

One thing to watch out for anytime you use mocks is to ensure you're really testing the algorithm in the class under test. It's not hard to accidentally end up directly testing the mocks, and accomplish no validation at all.

With everything rigged, it's time to *actually* write the test.

## Writing a test case

```
@Test
void addItemToEmptyCartShouldProduceOneCartItem() { ①
    Cart cart = inventoryService.addItemToCart("My Cart", 1); ②

    assertThat(cart.getCartItems()).extracting(CartItem::getQuantity)
    // ③
    .containsExactlyInAnyOrder(1);

    assertThat(cart.getCartItems()).extracting(CartItem::getItem) //
        .containsExactly(new Item(1, "TV tray", "Alf TV tray", 19.99));
    ④
}
```

- ① Go overboard on naming your test method to fully denote *what* it's testing.
- ② Exercise the InventoryService class's addItemToCart() method.
- ③ Use a plain old AssertJ assertion to extract each cart item's quantity. Verify there is only one cart item, and that its quantity is 1.
- ④ Use another AssertJ assertion to extract each cart item's Item, and verify there is only one, and it matches the initial data you entered in the setUp() method.

If that's a lot to take in, I recommend you go back and re-read the entire test case, line by line. There's a lot packed in there.

An important point of testing is to test not just the success path, but also the failure path. For example, what if the underlying ItemRepository or CartRepository were to fail. How would your code handle that?

# Running embedded container tests

So far, you've written a domain object test and a service test. That's a good start, but sometimes you need to test things on a grander scale. For example, verifying your web controller can properly interact with a backend service may have greater value.

To carry out this type of end-to-end test has historically required one of the following options:

- Stand up an expensive test bed and use a team of test engineers to manually exercise your app for every change. You also run the risk of your test procedures not keeping up with supported features.
- Write a slew of complex test cases that attempts to automate web pages, but falls apart when the slightest change occurs.

Either solution can turn out to be quite expensive and not necessarily build the confidence you seek.

Never fear, Spring Boot has a solution that may close the gap. Spring Boot's power to drive embedded web containers gives you the option to spin up an embedded container on a random port. Your test cases can now interact not with mocks or stubs, but instead a live version of your app.

To do that, look at this test case.

Full-blown web container for a test case

```

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT) ①
@AutoConfigureWebTestClient ②
public class LoadingWebSiteIntegrationTest {

    @Autowired WebTestClient client; ③

    @Test ④
    void test() {
        client.get().uri("/").exchange() //
            .expectStatus().isOk() //
            .expectHeader().contentTypeCompatibleWith(TEXT_HTML) //
            .expectBody(String.class) //
            .consumeWith(exchangeResult -> {
                assertThat(exchangeResult.getResponseBody()) //
                    .contains("<form method=\"post\" action=\"/add\"");
            });
    }
}

```

- ① `@SpringBootTest` asks Boot to spin up a working instance of your app. It seeks out the class decorated with `@SpringBootApplication`, and uses it to launch an embedded container. `WebEnvironment.RANDOM_PORT` is the option where a random port is selected.
- ② `@AutoConfigureWebTestClient` will create an instance of `WebTestClient` configured to talk to your app.
- ③ With autowiring, you can inject that instance of `WebTestClient` into the test case.
- ④ The actual test method uses the `WebTestClient` to invoke the home controller's / route. It includes the ability to apply some assertions. In this case, it verifies the HTTP response code, the Content-Type response header, and a Java 8 Consumer assertion against the response body.

Depending on the results, you can apply further verification. For example, with an HTML response, using something like

[jsoup](#) to parse and look for patterns make testing simple. For a JSON-based response, you can tap into JsonPath or JSONassert.

In *Chapter 6, Building APIs with Spring Boot*, you'll see how to verify hypermedia using things like the LinkDiscoverer API provided by Spring HATEOAS.

While powerful and easy to hook into, full-blown embedded container testing comes at a price. Here are some anecdotal stats on test performance.

*Table 4. Performance comparison of various test cases*

Test Name	Description	Time to Complete (ms)
ItemUnitTest	Domain object unit test	114ms
InventoryServiceUnitTest	Service-level unit test	147ms
LoadingWebSiteIntegrationTest	Embedded web container integration test	243ms

! This isn't a properly configured benchmark. Instead, it's an anecdotal observation from my IDE.

Spinning up embedded containers for test purposes is pricey. While it's tempting to jump in feet first thanks to Boot's amazing support, don't make it your starting point for testing.

It's better to build a comprehensive test strategy:

- Test your domain objects, *including* making sure they handle null properly.
- Test your services with collaborators properly mocked out. Properly exercise all your business logic.
- Create a handful of end-to-end smoke tests using embedded containers.

Be sure to handle null values properly. In this day and age, most modern IDEs handily support JSR-305. You can decorate your code with `@NonNull` and `@Nullable` annotations. This has become much more popular in the Java community compared to wrapping everything with `Optional`. Not only does `Optional` make your APIs more clunky, there is an overhead to all the boxing and unboxing.

Another debatable question, what's wrong with end-to-end testing? The table illustrates there is a performance cost, but that's not the only factor. Your tests also become more brittle when the scope increases. When you alter a domain object, you will definitely impact the corresponding unit test for it. There is a chance it will impact related service tests. But end-to-end test cases? Your changes, unless extensive, may go completely unnoticed.

However a change to a service is more likely to require updates not just to the immediate service-level test, but any end-to-end tests that cross that service. Writing too many large-scoped test cases can snowball into a lot more work everytime you change anything.

## Using Spring Boot's slice testing

So...you just found out about Spring Boot's awesome embedded container test support. And all your plans got crushed in the paragraphs where I cautioned against going overboard.

Is there anything in between unit and end-to-end integration testing?

Thankfully, the answer is yes: **slice testing**.

Spring Boot has the ability to switch on a subset of features. To give you a glance at what's available, check out this list:

- `@AutoConfigureRestDocs`
- `@DataJdbcTest`
- `@DataJpaTest`
- `@DataLdapTest`
- `@DataMongoTest`
- `@DataNeo4jTest`
- `@DataRedisTest`

- `@JdbcTest`
- `@JooqTest`
- `@JsonTest`
- `@RestClientTest`
- `@WebFluxTest`
- `@WebMvcTest`



All of the `@...Test` annotations come prebaked with the JUnit 5 `@ExtendWith(SpringExtension.class)` annotation, so you don't have to add it yourself.

Imagine wanting to write a JPA test using Boot's slice test feature. You'd write something like this.

#### JPA slice test case

```
@DataJpaTest ①
public class JpaSliceTest {

    @Autowired ItemRepository repository; ②

    @Test ③
    void itemRepositorySavesItems() {
        Item sampleItem = new Item( //
            "name", "description", 1.99);

        Item savedItem = repository.save(sampleItem);

        assertThat(savedItem.getId()).isNotNull();
    }
}
```

```
        assertThat(savedItem.getName()).isEqualTo("name");
        assertThat(savedItem.getDescription()).isEqualTo("description");
        assertThat(savedItem.getPrice()).isEqualTo(1.99);
    }
}
```

- ① `@DataJpaTest` switches on a subset of Spring Boot's features, focusing things on Spring Data JPA. It also enables JUnit 5 by automatically pulling in `@ExtendWith({SpringExtension.class})`.
- ② Autowire a Spring Data JPA `ItemRepository`.
- ③ Craft a test case using `ItemRepository`.

This JPA slice test case wires up everything related to Spring Data JPA, while ignoring other `@Component` bean definitions.

What's the impact?

This baby now runs in just 183ms. Compared to the 243ms for a full container, that's an improvement. *And* you get the benefit of testing against a real database (even if using embedded H2).

Your confidence should be boosted knowing there are no mocks to be found anywhere in this test case. Performance in this case involves shaving off about 25% of the spin-up cost!



Something that has existed for a *long* time, since the early days of the Spring Framework, is test performance enhancements. Buried in any Spring test that involves an application context, the framework will attempt to reuse existing application contexts. This can certainly speed up your test scenarios. That's why to emphasize the tradeoffs between unit, integration, and slice tests, I ran them individually in my IDE. Run them all together, and throughput will almost certainly be better. But it still makes for a nice spectrum of test options.

To round out testing, try creating one focused on Spring MVC controllers.

#### WebMvc slice test case

```
@WebMvcTest(HomeController.class) ①
public class HomeControllerSliceTest {

    private WebTestClient client;

    @MockBean ②
    InventoryService inventoryService;

    @BeforeEach
    void setUp(@Autowired MockMvc mockMvc) {
        this.client = MockMvcWebTestClient.bindTo(mockMvc).build(); ③
    }

    @Test
    void homepage() {
        when(inventoryService.getInventory()).thenReturn(Arrays.asList(
        //
        new Item(1, "name1", "desc1", 1.99), //
```

```

        new Item(2, "name2", "desc2", 9.99) //
    ));
when(inventoryService.getCart("My Cart")) //
    .thenReturn(Optional.of(new Cart("My Cart")));

client.get().uri("/").exchange() //
    .expectStatus().isOk() //
    .expectBody(String.class) //
    .consumeWith(exchangeResult -> {
        assertThat( //

exchangeResult.getResponseBody()).contains("action=/add/1\"");
        assertThat( //

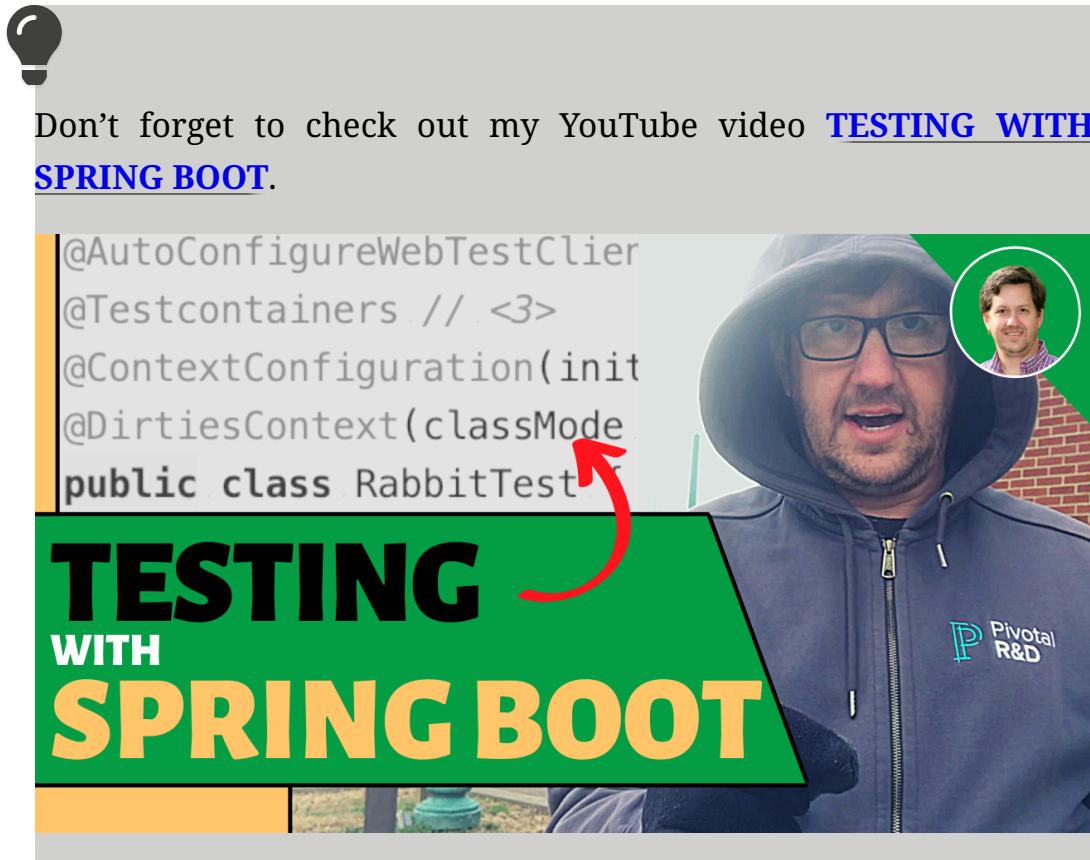
exchangeResult.getResponseBody()).contains("action=/add/2\"");
    });
}
}

```

- ① `@WebMvcTest(HomeController.class)` configures this test case using Spring Boot's WebMvc slice functionality *confined* to the `HomeController`.
- ② Mock out a collaborating `InventoryService` so you can focus on verifying the web controller, not the components underneath it.
- ③ Configure a `WebTestClient` using `MockMvcWebTestClient` and binding to the Mock MVC component created by `@WebMvcTest`.

The test method above uses all the same techniques already shown in earlier parts of this chapter. And when run on my machine, it finishes in 631ms.

With these options, you can pick and choose the test strategy that best serves your needs.



## Summary

In this chapter you:

- Tested domain objects.
- Tested services using mocks (courtesy of Spring Boot's `@MockBean`).
- Discovered how to run full container tests using Spring Boot.
- Found a way to test faster with slice-based testing using `@WebMvcTest` and `@DataJpaTest`.

In the next chapter, *Operations with Spring Boot*, you will discover how to add management support so after you deploy to production, you can handle Day 2 operations.

5

---

# OPERATIONS WITH SPRING BOOT

---

Spring Boot For The Win at work. Seems like a nice positive development, about to replace a few internal frameworks ("inner platforms").

~ Rainbow Coder @cowtowncoder

In the previous chapter, you learned how to test applications. You leveraged Spring Boot's slice-based testing and took advantage of Spring Boot's support for embeddable servlet container testing.

In this chapter, you're moving past development and into the realm of production. In production, you need to manage your applications and deal with Day 2 operational needs.

In this chapter, you are going to:

- Build an über JAR to deploy to production.

- Create a layer-based Dockerfile to bake a container.
- Bake a container without using a Dockerfile.
- Investigate using Spring Boot Actuator to operationalize your app.
- Decide which operational features are exposed and which ones are not.
- Add version details as application information.
- Customize the routes for your management services.

## Deploying your application to production

Josh Long has often referred to production as being the happiest place on Earth. That's because with Spring Boot, it's a snap to take all your hard efforts and, well, deploy them.

The next few sections will dive into various ways you can roll all your efforts into production.

### Going to production with an über JAR

*Chapter 1, Building a Web App with Spring Boot* briefly touched on this when it mentioned how Spring Boot's Maven plugin (and its Gradle plugin) will build your Java application into an executable JAR file.

And that's exactly true.

You've seen how `./mvnw spring-boot:run` is the way to run your Spring Boot application on the command line. And you can simply run the `main()` method from inside your IDE.

To assemble a runnable JAR file, execute this.

### Creating a runnable JAR file

```
$ ./mvnw package
```

Invoking Maven's package lifecycle will cause compilation, testing, and some other steps. The Spring Boot Maven plugin then uses this opportunity to assemble your über JAR.

### Creating an über JAR

```
$ ./mvnw package
[INFO] Scanning for projects...
[INFO]
...
[INFO] Building Hacking with Spring Boot - Chapter 5 - Operations
0.0.1-SNAPSHOT
...
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ hacking-spring-
boot-ch5-classic ---
[INFO] Building jar: /Users/gturnquist/personal/hacking-spring-boot-
code/5-classic/target/hacking-spring-boot-ch5-classic-0.0.1-
SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.4.4:repackage (repackage) @
hacking-spring-boot-ch5-classic ---
[INFO] Replacing main artifact with repackaged archive
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

```
[INFO] Total time: 1.709 s  
[INFO] Finished at: 2021-02-22T15:27:24-06:00  
[INFO] -----[
```



The output of this console has been trimmed down for readability.

There are two key things to notice:

- Building jar: /Users/gturnquist/personal/hacking-spring-boot-code/5-classic/target/hacking-spring-boot-ch5-classic-0.0.1-SNAPSHOT.jar
- Replacing main artifact with repackaged archive

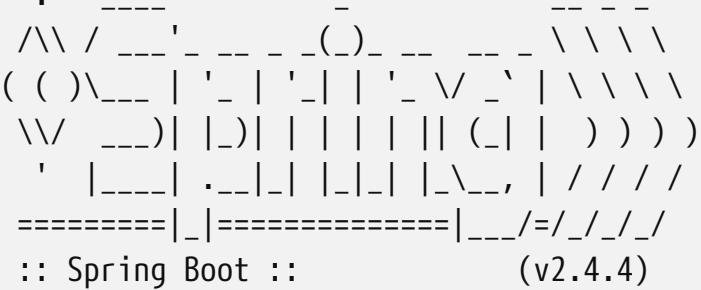
The first bullet point shows the `maven-jar-plugin` putting your compiled code into a JAR file in the `target` directory. This is Apache Maven's plugin, and it's doing what all good Java applications need.

But this isn't an executable JAR file. It's just your code.

That's where the second bullet point steps in. That comes from `spring-boot-maven-plugin`. It picks up the original JAR file, bundles it with dependencies and some special Spring Boot code, and forms a *new* JAR file to replace the original.

That is the executable JAR file. You can run it yourself.  
Running an executable JAR file

```
$ java -jar target/hacking-spring-boot-ch5-classic-0.0.1-SNAPSHOT.jar
```



```
2020-02-22 15:37:58.791 INFO 17346 --- [           main] c.g.h.r.HackingSpringBootApplication      : Starting HackingSpringBootApplication...
```

You can see it launching. Perfectly.

If you peek inside that JAR file (by typing `jar tvf target/hacking-spring-boot-ch5-classic-0.0.1-SNAPSHOT.jar`), you can see:

- Spring Boot's custom code to read the JAR file and load classes from nested JAR files.
  - Your application code.
  - All the 3rd-party libraries.



Did you know that Java's JAR spec doesn't support reading JAR files from inside another JAR file? It's the reason Spring Boot wrote custom code to load classes from a nested JAR.

Now take your JAR file and put it on a thumb drive. Or SSH to the remote box and upload it. All you need is a JDK to run it.

But what if you didn't even have Java on the target box? What then?

## Going to production with Docker

In the previous section, you crafted an über JAR. You ran it, and possibly trotted it out to production. It should work just fine.

As long as you have Java installed on your target machine. On ALL of your target machines.

But in this day and age, is keeping your production machines updated with the latest version of Java a burden for the operations team? They already track OS patches and everything else.

By using Docker, you can bundle up your application into a container that has Java baked right in.



In various parts of this book, you need to have Docker installed on your machine. While a big requirement five years ago, it's not such a big ask today. Leveraging it for production purposes and for testing purposes makes it a useful tool.

If you are new to Docker, you may wish to start [here](#). It has everything you need to install and take some tutorials.

Assuming you are warmed up to containers, a natural approach to wrapping a Java application into a container would be creating a `Dockerfile` like this.

Simplistic way to containerize your app

```
FROM adoptopenjdk/openjdk8:latest

ARG JAR_FILE=target/*.jar

COPY ${JAR_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

You can do really complex stuff when building a Docker container. This isn't one. In fact, this is about as simple as you can make things.

- It bases the container off a well-established OpenJDK provider.
- It pattern-matches the generated JAR file.
- It copies the JAR file into the container under a generic name.
- The launch point of the container starts the JAR file as if you were typing "java -jar" in the previous section.

The reason this is so simple is a real testimony to the ease of Spring Boot's über JAR support.

## So what's wrong?

Docker has a caching system built in. It uses layering to reduce time to build a container. Many parts of a Dockerfile can be captured into a layer, so that only the changes need to be applied when making an update.

Spring Boot's über JARs are based on combining your custom code with the managed dependencies tied to the version of Spring Boot you have selected. If all of that is mixed into a single layer, and if *anything* changes, the whole layer is invalidated.

Breaking up your application into multiple parts to support Docker's layer caching is a good approach. You could fumble around on stackoverflow.com to figure out how to configure your Dockerfile to do this.

But why not use the Spring Boot Maven plugin's built-in support for Docker to do all that for you?

To kick things off, you must tweak `spring-boot-maven-plugin` inside your `pom.xml` file, signaling you want a layered approach. Requesting a layered JAR in the build file

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

Look like the same old `spring-boot-maven-plugin` you always see? That's because it is.

In previous versions of Spring Boot, you had to add configuration settings to the plugin to enable layered support. But the Boot team actually flipped things so that layering is supported by default. You don't have to lift a finger!

With this in place, you use the same command to assemble your JAR file.

### Building a layered JAR

```
$ ./mvnw clean package
```

The code is very similar, but things are set up to support building a more optimal container. First of all, you can inspect the layers by doing this.

### Inspecting the new layers

```
$ java -Djarmode=layer-tools -jar target/hacking-spring-boot-ch5-classic-0.0.1-SNAPSHOT.jar list

dependencies
snapshot-dependencies
resources
application
```

The `jarmode=layer-tools` parameter is picked up by Spring Boot's layers code and when combined with the `list` command, prints out all the embedded layers.

That's nice. Now let's put this information to good use with a more sophisticated Dockerfile.

## Building a layer-based Dockerfile

```
FROM adoptopenjdk/openjdk8:latest as builder
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layer tools -jar application.jar extract

FROM adoptopenjdk/openjdk8:latest
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/spring-boot-loader/ ./
#COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/application/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

This is a multi-stage Docker build.

- Extracts the code into layers.
- Copies that code into different sections of the second container.
- Each COPY is subject to Docker's layer caching algorithm. Thus, 3rd-party libraries can be cached.
- Instead of running everything like a runnable JAR, it uses Spring Boot's custom launcher. No need to pay the cost of unpacking the JAR file on startup.

Depending on the version of all of your dependencies, you may or may not have a given layer. For example, if there are no snapshot dependencies, you can comment out the line that copies application/snapshot-dependencies (as shown above).

To build a container image, just do this.

### Containerizing your application

```
$ docker build . --tag hacking-with-spring-boot-classic

Sending build context to Docker daemon 42.17MB
Step 1/11 : FROM adoptopenjdk/openjdk8:latest as builder
--> 8d67cae7475e
Step 2/11 : WORKDIR application
--> Using cache
--> a523e2743633
Step 3/11 : ARG JAR_FILE=target/*.jar
--> Running in da20d93d8d27
Removing intermediate container da20d93d8d27
--> 133860631ada
Step 4/11 : COPY ${JAR_FILE} application.jar
--> f241e0039766
Step 5/11 : RUN java -Djarmode=layer-tools -jar application.jar
extract
--> Running in 041c261460d3
Removing intermediate container 041c261460d3
--> d75f9628df45
Step 6/11 : FROM adoptopenjdk/openjdk8:latest
--> 8d67cae7475e
Step 7/11 : WORKDIR application
--> Using cache
--> a523e2743633
Step 8/11 : COPY --from=builder application/dependencies/ ./
--> 61a5f7cd992c
Step 9/11 : COPY --from=builder application/spring-boot-loader/ ./
```

```
--> ebdc946c4bef
Step 10/11 : COPY --from=builder application/application/ ./
--> c099cdd1ff40
Step 11/11 : ENTRYPOINT ["java",
"org.springframework.boot.loader.JarLauncher"]
--> Running in adb9612d6b7b
Removing intermediate container adb9612d6b7b
--> 02a3819320a7
Successfully built 02a3819320a7
Successfully tagged hacking-with-spring-boot-classic:latest
```

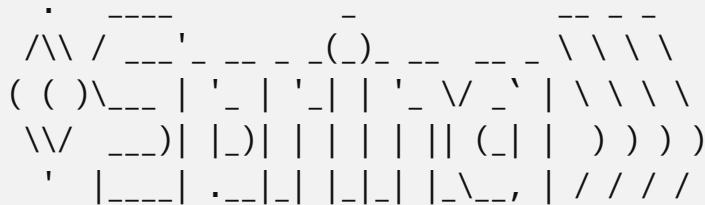
Congratulations! You just fashioned a Docker container image with a more optimal layout.

The first time around, most layers were built for the first time. Where it really starts to shine is when you make tweaks and adjustments to your source code. Run the Docker command again, and only the layers that really changed will get updated. The other stuff should stay the same.

And when you bump your project's Spring Boot version, other parts of the container will build as well.

Now why don't you take that container out for a spin?  
Running your Spring Boot application from Docker

```
$ docker run -it -p 8080:8080 hacking-with-spring-boot-classic:latest
```



```
=====|_|=====
 :: Spring Boot ::           (v2.4.4)

2020-02-23 22:42:28.758  INFO 1 --- [           main]
c.g.h.r.HackingSpringBootApplication : Starting
HackingSpringBootApplication on bbf124c6ced0 with PID 1
(/application/BOOT-INF/layers/application/classes started by root in
/application)
...

```

Hooray!

First time building a Docker image? If so, all of this may seem a bit daunting.

There is actually another way to build a container. Spring Boot has a built-in way to build a Docker container without writing a `Dockerfile`. At all!

Provided out-of-the-box is another option—using the `build-image` command in the Spring Boot Maven plugin. (The Spring Boot Gradle plugin also supports this.)

Building a Docker image with `spring-boot:build-image`

```
$ ./mvnw spring-boot:build-image
```

When you use `spring-boot:build-image`, Spring Boot will fetch a **buildpack** from the [Paketo Buildpacks project](#). From there, it will proceed to build a Docker container image...with no `Dockerfile`!

Spring Boot using Paketo to build a container image

```
[INFO] Building image 'docker.io/library/hacking-spring-boot-ch5-classic:0.0.1-SNAPSHOT'
[INFO]
[INFO] > Pulling builder image
'docker.io/paketobuildpacks/builder:base' 100%
[INFO] > Pulled builder image
'paketobuildpacks/builder@sha256:3618a2eae63a8c7b9125e0dc04b3f7beece1
49c4f1a98b542bdeac1c9ae19e07'
[INFO] > Pulling run image 'docker.io/paketobuildpacks/run:base-cnb'
100%
[INFO] > Pulled run image
'paketobuildpacks/run@sha256:6e230c0a716723b1f253f55ca21c87d42411f4f2
8cf1323d7deabd9e9b6bb94b'
[INFO] > Using build cache volume 'pack-cache-5a23c29cffea.build'
[INFO]
[INFO] [creator] Paketo Spring Boot Buildpack 3.5.0
[INFO] [creator] https://github.com/paketo-
buildpacks/spring-boot
[INFO] [creator] Creating slices from layers index
[INFO] [creator] dependencies
[INFO] [creator] spring-boot-loader
[INFO] [creator] snapshot-dependencies
[INFO] [creator] application
[INFO] [creator] Launch Helper: Contributing to layer
[INFO] [creator] Creating /layers/paketo-
buildpacks_spring-boot/helper/exec.d/spring-cloud-bindings
[INFO] [creator] Writing profile.d/helper
[INFO] [creator] Web Application Type: Contributing to
layer
[INFO] [creator] Servlet web application detected
[INFO] [creator] Writing
env.launch/BPL_JVM_THREAD_COUNT.default
[INFO] [creator] Spring Cloud Bindings 1.7.0: Contributing
to layer
[INFO] [creator] Downloading from
https://repo.spring.io/release/org/springframework/cloud/spring-
cloud-bindings/1.7.0/spring-cloud-bindings-1.7.0.jar
```

```
[INFO] [creator] Verifying checksum
[INFO] [creator] Copying to /layers/paketo-
buildpacks_spring-boot/spring-cloud-bindings
[INFO] [creator] 4 application slices
[INFO] [creator] Image labels:
[INFO] [creator] org.opencontainers.image.title
[INFO] [creator] org.opencontainers.image.version
[INFO] [creator] org.springframework.boot.spring-
configuration-metadata.json
[INFO] [creator] org.springframework.boot.version
[INFO]
[INFO] [creator] *** Images (2608d8db25be):
[INFO] [creator] docker.io/library/hacking-spring-boot-
ch5-classic:0.0.1-SNAPSHOT
[INFO]
[INFO] Successfully built image 'docker.io/library/hacking-spring-
boot-ch5-classic:0.0.1-SNAPSHOT'
```



A lot of this has been pruned for brevity!

In this example you manually baked the image by invoking the plugin.

It's possible to alter your build to bake an image everytime you do a build job. All it takes is adding an execution with a goal of `build-image` to the `spring-boot-maven-plugin` declaration in your `pom.xml`.

And with your newly baked *Paketo-based* image, you can test it out.

## Running your Paketo-built Docker image

```
$ docker run -it -p 8080:8080 docker.io/library/hacking-spring-boot-ch5-classic:0.0.1-SNAPSHOT

Setting Active Processor Count to 6
Calculating JVM memory based on 1073612K available memory
Calculated JVM Memory Configuration: -XX:MaxDirectMemorySize=10M -
-Xmx449742K -XX:MaxMetaspaceSize=111869K -
XX:ReservedCodeCacheSize=240M -Xss1M (Total Memory: 1073612K, Thread
Count: 250, Loaded Class Count: 17337, Headroom: 0%)
Adding 138 container CA certificates to JVM truststore
Spring Cloud Bindings Enabled
Picked up JAVA_TOOL_OPTIONS: -
Djava.security.properties=/layers/paketo-buildpacks_bellsoft-
liberica/java-security-properties/java-security.properties -
agentpath:/layers/paketo-buildpacks_bellsoft-
liberica/jvmkill/jvmkill-1.16.0-RELEASE.so=printHeapHistogram=1 -
XX:ActiveProcessorCount=6 -XX:MaxDirectMemorySize=10M -Xmx449742K -
XX:MaxMetaspaceSize=111869K -XX:ReservedCodeCacheSize=240M -Xss1M -
Dorg.springframework.cloud.bindings.boot.enable=true

      .----'--(_)--_ \ \ \
     (( )\__|'_|'_||'_\`|_\ )\ \
     \|/_||_|_|_| ||(_|_| )) )
      |____| .__|_|_|_|_|_\__,_| / / /
=====|_|=====|_|/_/=/\_/_/
 :: Spring Boot ::          (v2.4.4)

...
```

At this point, it looks pretty much like the other container. That's why the rest of its console output has been trimmed.

Each of these approaches has tradeoffs.

Method	Pros	Cons
Layer-based Dockerfile	<ul style="list-style-type: none"><li>You are in control by providing the Dockerfile.</li><li>With Spring Boot providing the layers, you can structure a premium container.</li><li>No need to throw away your hard won efforts. Spring Boot will support you where you stand.</li></ul>	<ul style="list-style-type: none"><li>You still have to maintain these containers.</li><li>If you don't bake your container properly, you may be using vulnerable layers.</li></ul>
Paketo Buildpack-based	<ul style="list-style-type: none"><li>No need to fuss with a Dockerfile.</li><li>Industry standards are baked into your container including upstream patches and SSL.</li><li>More time focused on development.</li></ul>	<ul style="list-style-type: none"><li>Your ability to alter Paketo's opinion is limited compared to a Dockerfile.</li></ul>

There are strengths for each, and Spring Boot is all about giving you the best options to choose.

At this stage, you can push your newly minted *containerized* application to someplace like [Docker Hub](#).



Much of the credit for this section belongs to Spring Boot lead Phil Webb's [blog article](#).

## Going to production with GraalVM

The next step in going to production is to optimize the startup time. With the beta release of Spring Native, releasing your Spring Boot application to [GraalVM](#) has never been easier!

**Spring Native** is the endeavor led by Sébastien Deleuze and Andy Clement with support from across the Spring team. Its aim is to make the entire Spring portfolio GraalVM-friendly.

The tl;dr on GraalVM is that it's a multi-language VM that targets Java, Python, and Ruby. By letting go of certain aspects, apps are able to run inside GraalVM instead of a classic Java VM and experience incredible startup speed and enhanced performance.

Things like reflection, bytecode manipulation, and proxies interfere with GraalVM, so the Spring team has been making changes to remove these impediments. At the same time, the Spring team has worked closely with the GraalVM team to make things more cohesive.

The result has been an amazing success! So far, you've been able to build apps in this book with Spring MVC and Spring Data. And with a few tweaks, that application can be built to run in sub-second time.

### Adding Spring Native to your pom.xml

```
<properties>
    <spring-native.version>0.9.1</spring-native.version>
</properties>
```

```
<dependency>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-native</artifactId>
    <version>${spring-native.version}</version>
</dependency>
```

Spring Native is currently an **experimental** library. At the time of writing, it's in beta release phase, being pre-1.0.

That's just the first piece of what's needed. While you could build locally for GraalVM, Spring Boot's ability to integrate with [Paketo Buildpacks](#) removes the need to install GraalVM on your machine. Instead, you can bake a Docker container on your machine that will *contain* GraalVM.

To do that, you have to alter `spring-boot-maven-plugin` as shown below:

Altering `spring-boot-maven-plugin` to support native image building

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <image>
            <builder>paketobuildpacks/builder:tiny</builder>
            <env>
                <BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
            </env>
        </image>
    </configuration>
</plugin>
```

Part of the native compilation requires activating the Just-In-Time (JIT) optimizer during the compilation phase. Normally the JIT activates when the code is run. This lets Java wait as long as possible before optimizing, making it possible to optimize on the target platform. The sacrifice includes start-up time among other things.

AOT is the concept of applying JIT during compilation.

Adding `spring-aot-maven-plugin` to `pom.xml`

```
<plugin>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-aot-maven-plugin</artifactId>
    <version>${spring-native.version}</version>
    <executions>
        <execution>
            <id>test-generate</id>
            <goals>
                <goal>test-generate</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

```
</execution>
<execution>
    <id>generate</id>
    <goals>
        <goal>generate</goal>
    </goals>
</execution>
</executions>
</plugin>
```

This plugin uses Spring Native to scour the code, looking for bits that need AOT application.

The details about *what* AOT involves is out-of-scope for this book. But if you're interested to learn more, visit the project.

Because you're using Spring Data JPA with Hibernate as the implementation of JPA, you also need to signal Hibernate to make some adjustments. This involves adding its own changes.

Adding `hibernate-enhance-maven-plugin` to `pom.xml`

```
<plugin>
    <groupId>org.hibernate.orm.tooling</groupId>
    <artifactId>hibernate-enhance-maven-plugin</artifactId>
    <version>${hibernate.version}</version>
    <executions>
        <execution>
            <id>enhance</id>
            <goals>
                <goal>enhance</goal>
            </goals>
            <configuration>
                <failOnErrors>true</failOnErrors>
```

```
<enableLazyInitialization>true</enableLazyInitialization>
    <enableDirtyTracking>true</enableDirtyTracking>

<enableAssociationManagement>true</enableAssociationManagement>

<enableExtendedEnhancement>false</enableExtendedEnhancement>
    </configuration>
</execution>
</executions>
</plugin>
```

And you're all set. And...that seems like a lot!

While it's possible to memorize all this, the secret sauce is to simply visit <https://start.spring.io>, enter all your dependencies, add **Spring Native**, and let the Spring Initializr add the extra plugin settings. You can either download a fresh project or simply click "Explore" and look at the build file to see what changes must be applied to your existing project. Once again, the Spring Initializr site is your ticket to building Spring Boot applications.



Some parts of the Spring portfolio don't (yet) support Spring Native. This includes Spring Boot's Actuator and DevTools modules. Spring Boot DevTools was covered earlier in *Chapter 3: Developer Tools for Spring Boot*. Spring Boot Actuator is covered further down in this chapter.

Assuming you've written all your code, the next step is to create an executable package.

## Building your native application

```
$ ./mvnw clean package

[INFO] Building Hacking with Spring Boot - Chapter 5b - Classic
Docker + GraalVM 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
-----

...
[INFO] --- hibernate-enhance-maven-plugin:5.4.29.Final:enhance
(enhance) @ hacking-spring-boot-ch5-classic-graalvm ---
[INFO] Starting Hibernate enhancement for classes on
/Users/gturnquist/personal/hacking-with-spring-boot-classic-code/5b-
classic/target/classes
Mar 20, 2021 11:33:56 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate ORM core version 5.4.29.Final

...
[INFO] --- spring-aot-maven-plugin:0.9.1-SNAPSHOT:generate (generate)
@ hacking-spring-boot-ch5-classic-graalvm ---
[INFO] Spring Native operating mode: native
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 15 source files to
/Users/gturnquist/personal/hacking-with-spring-boot-classic-code/5b-
classic/target/classes
[INFO] /Users/gturnquist/personal/hacking-with-spring-boot-classic-
code/5b-classic/target/generated-sources/spring-
aot/src/main/java/org/springframework/aot/StaticSpringFactories.java:
/Users/gturnquist/personal/hacking-with-spring-boot-classic-code/5b-
classic/target/generated-sources/spring-
aot/src/main/java/org/springframework/aot/StaticSpringFactories.java
```

```
uses or overrides a deprecated API.  
[INFO] /Users/gturnquist/personal/hacking-with-spring-boot-classic-  
code/5b-classic/target/generated-sources/spring-  
aot/src/main/java/org/springframework/aot/StaticSpringFactories.java:  
Recompile with -Xlint:deprecation for details.  
[INFO] /Users/gturnquist/personal/hacking-with-spring-boot-classic-  
code/5b-classic/target/generated-sources/spring-  
aot/src/main/java/org/springframework/aot/StaticSpringFactories.java:  
Some input files use unchecked or unsafe operations.  
[INFO] /Users/gturnquist/personal/hacking-with-spring-boot-classic-  
code/5b-classic/target/generated-sources/spring-  
aot/src/main/java/org/springframework/aot/StaticSpringFactories.java:  
Recompile with -Xlint:unchecked for details.  
[INFO] Using 'UTF-8' encoding to copy filtered resources.  
[INFO] Using 'UTF-8' encoding to copy filtered properties files.  
[INFO] Copying 4 resources  
[INFO]  
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ hacking-spring-  
boot-ch5-classic-graalvm ---  
[INFO] Building jar: /Users/gturnquist/personal/hacking-with-spring-  
boot-classic-code/5b-classic/target/hacking-spring-boot-ch5-classic-  
graalvm-0.0.1-SNAPSHOT.jar  
[INFO]  
[INFO] --- spring-boot-maven-plugin:2.4.4:repackage (repackage) @  
hacking-spring-boot-ch5-classic-graalvm ---  
[INFO] Replacing main artifact with repackaged archive  
[INFO] -----  
-----  
[INFO] BUILD SUCCESS  
[INFO] -----  
-----  
[INFO] Total time: 10.020 s  
[INFO] Finished at: 2021-03-20T23:34:05-05:00  
[INFO] -----  
-----
```

You may see what appears to be errors from the `spring-aot-maven-plugin`, but they simply indicate strategies the plugin decides aren't needed based on the dependencies you've selected.



If you are using Spring Native and you want to run the app before baking anything, then `./mvnw clean spring-boot:run` is going to cause a little trouble. That's because `spring-boot:run` never hits the package phase. The recommended way to run things with Spring Native is `./mvnw clean package spring-boot:run`. That will clean out old stuff, run the AOT processing, and *then* execute the code.

Assuming everything is built, you're ready to bake a Docker container with GraalVM and your app inside it.

### Baking a Docker + GraalVM container

```
$ ./mvnw clean spring-boot:build-image  
...pruned for brevity...  
  
[INFO] [creator] ==> EXPORTING  
[INFO] [creator] Adding 1/1 app layer(s)  
[INFO] [creator] Reusing layer 'launcher'  
[INFO] [creator] Reusing layer 'config'  
[INFO] [creator] Reusing layer 'process-types'  
[INFO] [creator] Adding label  
'io.buildpacks.lifecycle.metadata'  
[INFO] [creator] Adding label 'io.buildpacks.build.metadata'  
[INFO] [creator] Adding label  
'io.buildpacks.project.metadata'
```

```
[INFO] [creator] Adding label  
'org.opencontainers.image.title'  
[INFO] [creator] Adding label  
'org.opencontainers.image.version'  
[INFO] [creator] Adding label  
'org.springframework.boot.spring-configuration-metadata.json'  
[INFO] [creator] Adding label  
'org.springframework.boot.version'  
[INFO] [creator] Setting default process type 'web'  
[INFO] [creator] *** Images (5f6688062a56):  
[INFO] [creator] docker.io/library/hacking-spring-boot-  
ch5-classic-graalvm:0.0.1-SNAPSHOT  
[INFO] [creator] Reusing cache layer 'paketo-  
buildpacks/graalvm:jdk'  
[INFO] [creator] Adding cache layer 'paketo-  
buildpacks/native-image:native-image'  
[INFO]  
[INFO] Successfully built image 'docker.io/library/hacking-spring-  
boot-ch5-classic-graalvm:0.0.1-SNAPSHOT'
```



This step can take awhile, sometimes up to fifteen minutes. In fact, the Spring Native team recommends you run your Docker engine with at least 8 GB of memory. On some platforms, Docker runs at a default of 2 GB.

That previous output is heavily shortened to fit this book. You can see pages of activity. And depending on your machine, I've seen it take up to fifteen minutes to bake a Docker + GraalVM image. So don't be surprised if it takes awhile.

The gold that is buried in that huge console output is the fact that your applications has been compiled, optimized, and baked into a Docker container. And you can run it!

# Running a Dockerized GraalVM app

```
$ docker run -p 8080:8080 docker.io/library/hacking-spring-boot-ch5-classic-graalvm:0.0.1-SNAPSHOT
```

2021-03-21 04:20:19.425 : This application is bootstrapped with code generated with Spring AOT

2021-03-21 04:20:19.636 : Tomcat started on port(s): 8080 (http) with contextPath: ''

2021-03-21 04:20:19.638 : Started HackingSpringBootApplication in 0.226 seconds (JVM running for 0.23)

Run this container, exposing port 8080. And wow, check out that start time! 0.226 seconds.

Let me repeat that—the startup time on that app was **0.226 seconds**.

To top things off, you also have a Docker container that you can manage with whatever solution you like. Push it to Docker Hub, deploy to your Kubernetes cluster, or slap together a bash script.

Paketo Buildpacks combined with Spring Native and its plugins a la Spring Initializr provide an incredible experience. Instead of stitching together information from a dozen different blog posts and running various esoteric commands to manipulate Java and Docker, these tools show up to get the job done for you.

Leaving you the ability to focus on writing code that solves problems.

So what *exactly* is the benefit of using GraalVM? Is sub-second startup time that compelling?

Yes..and no.

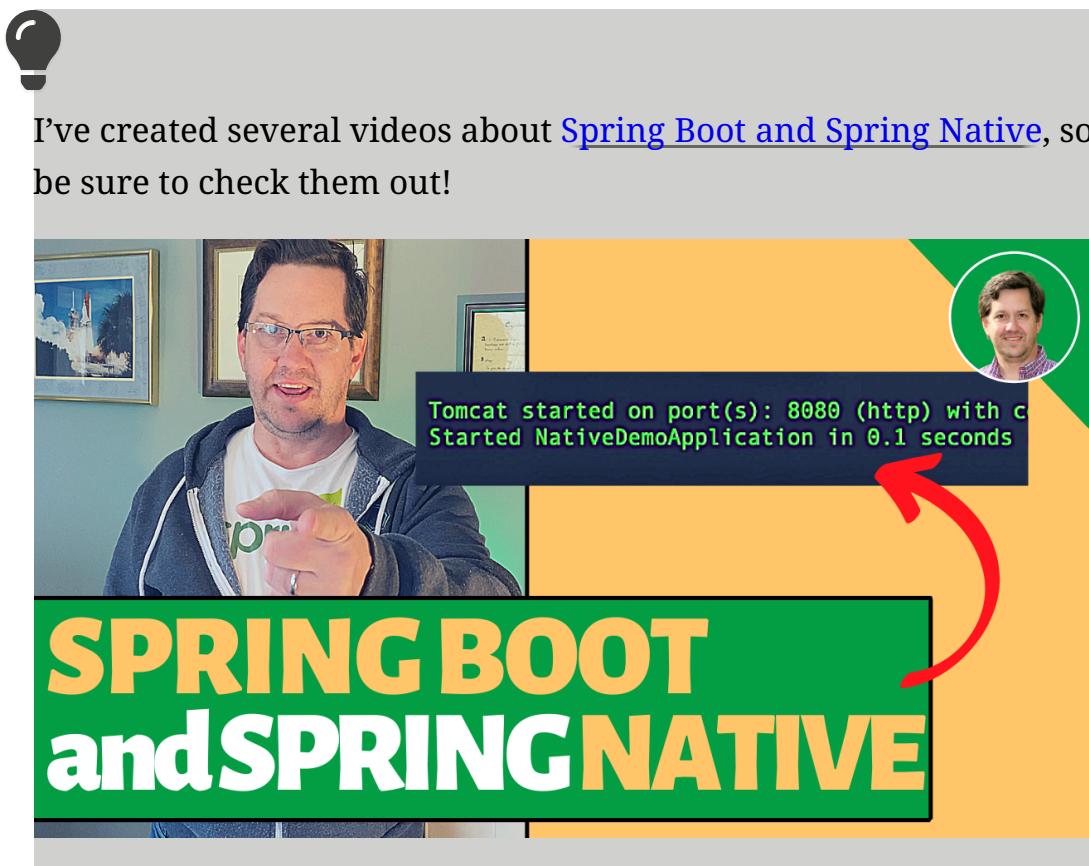
If you're running a single-node app that stays up all the time and only making changes every other week, perhaps not.

But if you're running 10,000 nodes and makes deploying changes daily, then you should entertain. The scale of saving time can result in significant savings on your cloud bill.

If you're running in a serverless setup and trying to avoid having to move from Java to Go because of startup time, then

this is probably exactly what you need.

Just remember that building apps for GraalVM takes a lot longer. If that cost to perform AOT compilation results in scalable savings, then you should entertain it.



You can choose to push über JARs or Docker containers to production. You can even automate the whole process with Jenkins, Concourse, or some other CI solution. Whatever you prefer, I highly recommend automating the process. Being able to push each change to production will seriously amortize both the cost and risk.

# Managing your application in production

You finished Day 1 by deploying your application to production. Now it's time for Day 2, managing your application.

The rest of this chapter will show the various tools provided by Spring Boot's Actuator module to help monitor your application.

## Pinging your app with Spring Boot Actuator and /actuator/health

So you've built a Spring Boot application. You crafted some web pages and render them using WebFlux. You stirred in a little Spring Data. Spring Boot's developer tools sure did speed up the cycles to get that hammered out.

And five minutes ago...you pushed that runnable JAR file to the server (or perhaps installed from a thumb drive in your air-gapped server room).

So now what?

The first question from the Ops team now managing your app will probably be, "can we ping it?" And they're just getting warmed up. What about metrics? Stats? Detailed health checks?

Does it plays nicely with the database they *also* have to manage?

You could start to craft some extra controllers to meet these requests.

Or...you could add **Spring Boot Actuator** and call it a day.

To add operational support to your Spring Boot app, add this to your build file.

Adding Spring Boot Actuator to the build

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

Relaunch your application, and a new series of components have been added to your application.

At first, you can see this tiny message in the console's log output.

Spring Boot Actuator reported in console log

```
2020-05-24 23:06:38.310  INFO 57200 --- [ restartedMain]
o.s.b.a.e.web.EndpointLinksResolver      : Exposing 2 endpoint(s)
beneath base path '/actuator'
```

The message above shows that Spring Boot Actuator is live, and that two endpoints are activated. It does NOT tell *what* has been activated. That's partly because Spring Boot (and Spring

Framework) have taken a more proactive stance on security and show less specific details like web routes (unless you dial up the log levels yourself).



Back in *Chapter 1, Building a Web App with Spring Boot*, when you created Spring MVC controllers, the actual routes you defined did NOT appear in the console logs. If you used Spring Boot 2.1 or earlier, you may have gone looking for that output. It was removed in a later version.

Remember the first request from the Ops team to ping your app? With a slight grin on your face, tell them to hit <http://localhost:8080/actuator/health>, where they can expect to see something like this.

#### Barebones health check

```
{  
  status: "UP"  
}
```

First Ops request...done!

If this "UP" response doesn't appear *that* impressive, just consider the fact that you didn't have to write it. Why manage code for something not directly tied to your business needs. Instead, by picking up Spring Boot Actuator, the Boot team can manage it for you.



Before going any deeper, it's most valuable to have a JSON viewer installed in your browser. The JSON served by Spring Boot Actuator is in a compact format that's frankly not built solely for human consumption. Using a tool like [JSONViewer for Chrome](#) or [JSONView for Firefox](#) are good candidates.

When it comes to operational maintenance, you're just getting started. It's possible to show more details, if you add the following settings to `src/main/resources/application.properties`. Configuring your app to return detailed health status

```
management.endpoint.health.show-details=always
```

Restart your application then revisit <http://localhost:8080/actuator/health>. You'll get something more detailed.

#### Detailed health check

```
{
  status: "UP",
  components: {
    db: {
      status: "UP",
      details: {
        database: "H2",
        validationQuery: "isValid()"
      }
    },
    diskSpace: {
```

```
        status: "UP",
        details: {
            total: 499963174912,
            free: 39421014016,
            threshold: 10485760,
            exists: true
        }
    },
    ping: {
        status: "UP"
    }
}
}
```

This detailed health check is automatically assembled by Spring Boot Actuator using it's knowledge of autoconfiguration. In this case, it shows:

- H2 details (health status and the query used to verify H2)
- Disk consumption (health status and metrics)
- Ping status

If you activate other components (e.g. Redis, Apache Cassandra, various relational data stores, RabbitMQ message broker, email support, or other supported modules), then Spring Boot Actuator will *automatically* register matching HealthIndicator implementations. Each one will provide a health status of UP, DOWN, OUT\_OF\_SERVICE, or UNKNOWN.

All health indicators are added up for a top-level status (which is what's given by default).



Later, in *Chapter 8, Securing Your Application with Spring Boot*, you'll learn how to control who gets access to detailed health reports.

## Serving valuable app details with /actuator/info

The health of your application isn't the only maintenance data that's needed. Valuable information to include in deployed applications are the versions of various components you used!

Imagine getting a call at 3:00 a.m. from one of your customers about something being broken. You spend hours combing tickets and commit logs, only to discover you fixed that exact problem...two week ago. Looks like your customer is using an old version.

This is something that could have been solved up front with a quick check of what's actually deployed. With Spring Boot Actuator, you can embed details of your build file into your `application.properties` with just a few lines.

### Injecting Maven build details

```
info.project.version=@project.version@  
info.java.version=@java.version@  
info.spring.framework.version=@spring-framework.version@  
info.spring.data.version=@spring-data-bom.version@
```

When using Maven, these tokens will automatically get injected with the values found in the build file.

The other thing you can include, if you're using git as your version control mechanism, is automatic access to version control information.

Add this plugin to `pom.xml`, right below `spring-boot-maven-plugin`.

Adding git information

```
<plugin>
    <groupId>pl.project13.maven</groupId>
    <artifactId>git-commit-id-plugin</artifactId>
</plugin>
```

The `git-commit-id-plugin` is used by Maven to create a `git.properties` file, which Spring Boot will automatically pick up.

Putting it all together, you can get a very detailed status.

Launching the app and visiting <http://localhost:8080/actuator/info> will show something like this.  
Showing library versions and source version details

```
{
  "project": {
    "version": "0.0.1-SNAPSHOT"
  },
  "java": {
```

```
        "version": "1.8.0_242"
    },
    "spring": {
        "framework": {
            "version": "5.3.3"
        },
        "data": {
            "version": "2020.0.3"
        }
    },
    "git": {
        "commit": {
            "time": "2021-01-25T23:33:39Z",
            "id": "f780a4e"
        },
        "branch": "master"
    }
}
```

With this type of information available, you can quickly glean if your customer is running an up-to-date version. If not, you can advise them to upgrade. If they are on the latest supported version, you can proceed to finding a solution much faster.

Asking Maven to inject data from the build file as well as version control information allows seamless integration between what you're building and what is getting shipped to customers. By avoiding manual intervention, you greatly reduce the risk of having out-of-date details served to your users.



Not only can you configure information through `application.properties`, you can also implement Spring Boot's `InfoContributor` interface. Implement it and register it as a bean, and Spring Boot will show whatever you want to serve. It's a programmatic way to hook in information.

## Accessing additional actuator endpoints

Time to step off a ledge and into the deep.

Until now, you've only seen the *default* actuator endpoints, `health` and `info`, exposed to the web. Spring Boot actually has many other endpoints, but they are disabled from web access by *default*.



Java Management Extensions (JMX) actuator endpoints are *enabled* by default, while their web-based counterparts are *disabled* by default. This is primarily for security reasons. JMX in its default configuration requires you to run JConsole on the same machine.

Web access to actuator endpoints are a complete opposite. While convenient, it becomes super easy to expose all your application has to offer. This is why Spring Boot 2.4 *disables* web access to actuator endpoints. The only exception being a shallow health check (only reporting UP) as well as info data (information you must choose to include).

So...you can include everything if you really wanted to. All it takes is a single line in `application.properties`.

Exposing all web-based actuator endpoints

```
management.endpoints.web.exposure.include=*
```

This single line will expose all endpoints. Simple, but **not** recommended. Let me repeat—you do NOT want to do this.

Even if you agree to all the exposed details, you have no way of knowing that a future actuator endpoint, whether released by the Boot team or a custom endpoint of your own won't accidentally expose dangerous details.

No, a better approach is to *explicitly* list everything you want to expose.

## Exposing all web-based actuator endpoints with an explicit list

```
management.endpoints.web.exposure.include=auditevents,beans,caches,conditions,configprops,env,flyway,health,heapdump,httptrace,info,logfile,loggers,metrics,mappings,shutdown,threaddump
```

It may appear a bit crammed, but it really is better to spell out exactly what you wish exposed. When you restart your application, you can expect to see something like this in the console logs.

Additional exposed endpoints in console logs

```
2020-05-28 21:29:22.577 INFO 89444 --- [ restartedMain]
o.s.b.a.e.web.EndpointLinksResolver      : Exposing 12 endpoint(s)
beneath base path '/actuator'
```

Now how about exploring these actuators?

## Accessing loggers with /actuator/loggers

With loggers exposed, you can list all the various loggers and their levels.

Shortened excerpt of loggers

```
{
  levels: [
    "OFF",
    "ERROR",
    "WARN",
    "INFO",
```

```

    "DEBUG",
    "TRACE",
],
loggers: {
  ROOT: {
    configuredLevel: "INFO", ①
    effectiveLevel: "INFO", ②
  },
  com: {
    configuredLevel: null, ③
    effectiveLevel: "INFO", ④
  },
  com.greglturnquist: {
    configuredLevel: null,
    effectiveLevel: "INFO",
  },
  com.greglturnquist.hackingspringboot: {
    configuredLevel: null,
    effectiveLevel: "INFO",
  },
...

```

- ① The ROOT logger is configured (by Spring Boot itself) to the INFO level.
- ② "Effective" level is INFO, since no other policy overrides it.
- ③ The application's top-level, com, is NOT configured with a log level.
- ④ The derived log level for this package is INFO.

Since every package loaded into the system is assessed and split up into sections, there are entries for:

- com
- com.greglturnquist
- com.greglturnquist.hackingspringboot

- and more!

While a bit detailed, it clearly indicates where you have the power to set logging levels.

Glancing at the levels supported at the top, you could quickly configure `com` to level TRACE if you wish (and probably bury yourself in console logs!).

In fact, why not try just that!

Spring Boot's actuators don't just report data. Some accept inputs.

Changing a single logger's level

```
$ curl -v -H 'Content-Type: application/json' -d '{"configuredLevel": "TRACE"}' http://localhost:8080/actuator/loggers/com.greglturnquist/  
* Trying ::1...  
* TCP_NODELAY set  
* Connected to localhost (::1) port 8080 (#0)  
> POST /actuator/loggers/com.greglturnquist/ HTTP/1.1  
> Host: localhost:8080  
> User-Agent: curl/7.54.0  
> Accept: */*  
> Content-Type: application/json  
> Content-Length: 28  
>  
* upload completely sent off: 28 out of 28 bytes  
< HTTP/1.1 204 No Content  
<  
* Connection #0 to host localhost left intact
```

- The URI to make changes is `/actuator/loggers/{package}`, where `{package}` is the key shown in the JSON document shown earlier in this section.
- The content must be `application/json`.
- The provided data must be a `configuredLevel` set to a valid log level.

If you ping that logger (<http://localhost:8080/actuator/loggers/com.gregturnquist/>), then expect to see this.

Inspecting a single logger after changing it

```
{  
  configuredLevel: "TRACE",  
  effectiveLevel: "TRACE",  
}
```

If you wish to clear out a logger, simply swap out a log level with `null`.

Clearing a logger's level

```
$ curl -v -H 'Content-Type: application/json' -d '{"configuredLevel":  
null}' http://localhost:8080/actuator/loggers/com.gregturnquist/
```

Ping it again (<http://localhost:8080/actuator/loggers/com.gregturnquist/>) and verify things are properly restored.  
A logger returned to original condition

```
{  
    configuredLevel: null,  
    effectiveLevel: "INFO",  
}
```

This is useful if you want to alter the logger settings and then observe the results. For example, debugging a current scenario based on operational data flowing into the system.

What this is not so good for, is pushing out general configuration settings. If you're in a cloud-based environment, each change you make will only alter a single instance. And the changes won't last. So the next time you either restart an instance or push out a new version, your changes are gone.

## Reading operational data

There is a lot of operational data you can examine. The following subsections will provide various endpoints that offer different slices.

### Reading up on threads with /actuator/threaddump

Visit <http://localhost:8080/actuator/threaddump>, and you'll see every single thread running in your application. Heads up—it's a lot.

When it comes to server-side threads, there are lots of different groupings. Apache Tomcat runs with a thread pool. The

LiveReload server provided by **Spring Boot DevTools** runs in a separate thread.

You can take snapshots (or write a tool?) to inspect things at various phases of your application. What do you see when it's under load? When idle?

## Analyzing data using /actuator/heapdump

Go ahead and visit <http://localhost:8080/actuator/heapdump>. Instead of seeing some squashed JSON (or prettified via a formatter), your browser will instead download a GZip-compressed **hprof** file.

Do this to access it.

### Viewing a heapdump file

```
$ jhat ~/Downloads/heapdump
Reading from /Users/gturnquist/Downloads/heapdump...
Dump file created Fri May 31 22:47:11 CDT 2030
Snapshot read, resolving...
Resolving 354920 objects...
Chasing references, expect 70
dots.....
.....
Eliminating duplicate
references.....
.....
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

 If your downloaded file has a ".dms" or any other file extension, jhat won't read it. Remove the extension and then you can investigate as shown in this chapter.

You can now view the heapdump file generated by Spring Boot Actuator's ThreadDumpEndpoint by opening a tab to <http://localhost:7000>.

You can sift through it. At the bottom, are several convenient reports including:

- Show heap histogram
- Show instance counts for all classes (including platform)
- Show instance counts for all classes (excluding platform)

Even though this set of reports is quite rudimentary, because jhat comes with the JDK, it's available everywhere.

However, should you desire to do a more serious analysis, I'd recommend installing something like [Visual VM](#).

If you happen to use [sdkman](#) to manage your instances of Java, getting Visual VM is simple.

```
$ sdk list visualvm  
$ sdk install visualvm 2.0.6  
$ visualvm --jdkhome $JAVA_HOME
```

After Visual VM is started, do the following steps:

1. Click File → Load.
2. Navigate to the folder heapdump was downloaded.
3. Select the file and open it.

From there, you can visit various parts of the file in a much more friendly way than jhat could.

## Tracing HTTP calls with /actuator/httptrace

Spring Boot Actuator provides a convenient way to see who's making calls to your app. Along with that comes a whole host of questions.

- What types of clients are most popular? Mobile? A certain browser?
- Are people making requests in certain languages? German? English? Do we need internationalization?
- Which endpoints are being hit the most?
- From where?

This is the type of data analysis that can help you tailor your site to match customer needs. And to switch it on is a snap!

Spring Boot provides the `HttpTraceRepository` interface. Any registered bean that implements this interface will be picked up and automatically used to service calls to /actuator/httptrace.

And if you didn't get the hint, "any registered bean" means that Spring Boot does not automatically provide that type of bean. Instead, you must make the choice and create one.

To get familiar with this endpoint, you can use an out-of-the-box memory-based version of this interface—the `InMemoryHttpTraceRepository`.

You can easily add this bean definition to the class where you defined the Spring Boot application runner.

Registering an in-memory HTTP trace repository inside  
HackingSpringBootApplication

```
@Bean ①
HttpTraceRepository traceRepository() { ②
    return new InMemoryHttpTraceRepository(); ③
}
```

This bit of code uses standard Spring Framework bean registration tactics:

- ① `@Bean` will register the returned object as a Spring bean in the application context.
- ② The method will return a bean of type `HttpTraceRepository`.
- ③ The method actually creates an instance of Spring Boot Actuator's `InMemoryHttpTraceRepository`, as you'll explore further.

That code is all that's needed to activate HTTP tracing!

## 1. Restart the application

2. Visit <http://localhost:8080>.

3. Click around on some of the links.

With a bean of type `HttpTraceRepository` registered in the application context, Spring Boot Actuator will discover it and activate its `HttpTraceEndpoint` automatically.

And when the endpoint is activated, it will hook itself into Spring MVC, sniff out every web call, and log it.

Assuming you've accumulated some clicks, visit <http://localhost:8080/actuator/httptrace> and see the results. You can find various interactions, including this instance of adding an item to the cart.

Tracing the addition of an item to the cart

```
{
  traces: [
    {
      timestamp: "2020-06-01T14:26:15.692Z",
      principal: null,
      session: null,
      request: {
        method: "POST",
        uri: "http://localhost:8080/add/1",
        headers: {
          Cookie: [
            "_ga=GA1.1.1546091496.1540230553; Idea-e47d3d38=5f55a667-ce22-497b-b1ee-1b9581ed7312"
          ],
          Accept: [

```

```
"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3"
],
Upgrade-Insecure-Requests: [
    "1"
],
Connection: [
    "keep-alive"
],
Referer: [
    "http://localhost:8080/"
],
User-Agent: [
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169 Safari/537.36"
],
Host: [
    "localhost:8080"
],
Accept-Language: [
    "en-US,en;q=0.9"
],
Accept-Encoding: [
    "gzip, deflate, br"
]
},
remoteAddress: null
},
response: {
    status: 303,
    headers: {
        transfer-encoding: [
            "chunked"
        ],
        Location: [
            "/"
        ]
    }
}
```

```
        ]
    }
},
timeTaken: 179
},
...
}
```

This trace record includes a whole slew of information:

- Timestamp
- Security details (if provided)
- Session id
- Request details (HTTP method, URI, headers)
- Response details (HTTP status code, headers)
- Total time to process (milliseconds)

If you'll remember, this trace was produced by the `InMemoryTraceRepository` bean you registered. Being a memory-based repository means two key things:

- The trace records will only exist in the current instance. If you run multiple copies behind a load balancer, each copy will have its own data.
- Restarting an instance will throw away all results.

Need to take things to next level?

This level of tracing is alright for small stuff. But if you need real tracing and observability, then you should investigate Spring Cloud Zipkin or Spring Cloud Sleuth.

Josh Long's blog post [Distributed Tracing with Spring Cloud Sleuth and Spring Cloud Zipkin](#) is a good introduction to the benefits of this distributed, scalable solution that provides deeper insight into what is happening your application.

There is also a great video about [Zipkin and Distributed Tracing](#) on the official Spring team's YouTube channel.

Because this is *Hacking with Spring Boot*, and not *Hacking with Spring Cloud\_*, we'll wrap things up here.

## Other operational readouts

There are more operational readouts available.

Actuator Endpoint	Description
/actuator/auditevents	Exposes audit events.
/actuator/beans	Reports all the beans configured in the application context (including yours as well as the ones autoconfigured by Boot).
/actuator/caches	List all caches.

<b>Actuator Endpoint</b>	<b>Description</b>
/actuator/conditions	Reports what Spring Boot did and didn't autoconfigure and why.
/actuator/configprops	Exposes all configuration properties.
/actuator/env	Reports on the current system environment.
/actuator/flyway	Show any Flyway database migrations that have been applied.
/actuator/mappings	Details about all Spring WebFlux routes.
/actuator/metrics	Gather and serve metrics using Micrometer.

They aren't complex and provide straight forward data, i.e. there's not a lot to be gained by expanding upon them here.

## Customizing management service routes

While it's nice to use the toolkit's default values, sometimes you need to bend your application to the sys admin's monitoring agents. That's why Spring Boot Actuator provides the means to tune endpoints.

By adding the following setting to `src/main/resources/application.properties`, you can tune the path for actuator.

#### Adjusting actuator's root endpoint

```
management.endpoints.web.base-path=/manage
```

This will move all Actuator endpoints from `/actuator` to `/manage`.

Moving from `/actuator/loggers`, to just `/logs` involves a different change.

#### Remapping an actuator's path

```
management.endpoints.web.base-path=/  
management.endpoints.web.path-mapping.loggers=logs
```

This alters Spring Boot Actuator's base path to `/` and then changes the name for loggers.

You can apply similar changes to all the other Actuator endpoints.

## Summary

In this chapter you:

- Built an executable JAR file.
- Created a layer-based Dockerfile and baked a container.

- Used Paketo Buildpacks and baked a container with no Dockerfile.
- Added Spring Boot Actuator.
- Exposed only the management services that were needed.
- Customized application info with build information.
- Customized the routes to your management services.

In the next chapter, *Building APIs with Spring Boot*, you will learn how to build APIs, augment them with hypermedia, and document your application in an efficient and effective way.

6

---

# BUILDING APIs WITH SPRING BOOT

---

You can't go far wrong with Spring. Take a look at Spring Boot and build some REST api backends that you can call from a frontend  
React, Vue or Angular app

~ Kevin Hooke @kevinhooke

In the previous chapter, you learned how to use Spring Boot Actuator in an operational setting.

Now it's time to build your interfaces for external systems—APIs.

In this chapter you'll:

- Learn how to build JSON web services
- Write self-documenting tests using Spring REST Docs.

- Register these various API interactions in an API portal served automatically by Spring Boot.
- Craft hypermedia representations using Spring HATEOAS.
- Expand the API portal to include hypermedia links.

## Creating an HTTP web service

The simplest API is the one that you query and it hands you back data. In the olden days, it was common to serve up either XML or some binary data based on the technology stack.

In today's world of e-commerce and mashups, the name of the game is JSON (JavaScript Object Notation). And creating a Spring MVC endpoint that gives you JSON instead of HTML couldn't be easier.

In the previous chapters, you developed a shopping Cart containing one or more Item objects. You then persisted these objects into H2 using JPA and displayed it to the user using templates.

Imagine now, instead of fetching data from your repository, building up a model, and binding it to a template...you just return the data itself?

Defining an API controller

```
@RestController ①
public class ApiItemController {
```

```
private final ItemRepository repository; ②

public ApiItemController(ItemRepository repository) {
    this.repository = repository; ③
}
```

- ① `@RestController` indicates that this is a Spring web controller, but instead of returning HTML, the return value is written directly into the response body.
- ② This controller has a copy of the same `ItemRepository` built in previous chapters.
- ③ It's injected by **constructor injection**.

Let me repeat, because it's important—a class flagged with `@RestController` is just like a class marked with `@Controller` in the sense that Spring Boot will automatically find it and register it via **component scanning**. `ApiItemController` will be picked up *automagically* and registered as a Spring bean.

But that's not all. When a web call is made, and the method returns a value, that value isn't used to render HTML. Instead, the returned object is serialized and then written directly into the response body.

To get a clearer picture, start drafting a web method to return ALL `Item` objects stored in your inventory.  
Fetching all the data for your API

```
@GetMapping("/api/items") ①  
Iterable<Item> findAll() { ②  
    return this.repository.findAll(); ③  
}
```

- ① The `@GetMapping` annotation signals that this method will respond to HTTP `GET /api/items` calls.
- ② The method returns `Iterable<Item>` meaning zero or more `Item` objects will get serialized into a JSON structure and written to the response body.
- ③ This web method delegates to your Spring Data `ItemRepository`, exercising its `findAll()` method, which itself returns `Iterable<Item>`.

This one is pretty simple. You want to get a collection of `Item` objects. No filtering. No processing. No transformation of any kind. Essentially a passsthrough to JPA.

The first thing leaping off the page should be the fact this returns a collection—an `Iterable`.

! Just because you *can* doesn't mean you *should*. This method has the potential to return one `Item` object—or 100,000. For demonstration purposes, this is adequate. But in a real world system, you may consider using Spring Data's paging to constrain a maximum.

To fetch a single `Item` object isn't hard either.

## Fetching a single `Item` for your API

```
@GetMapping("/api/items/{id}") ①
Optional<Item> findOne(@PathVariable Integer id) { ②
    return this.repository.findById(id); ③
}
```

- ① This `@GetMapping` annotation shows this method ready to handle GET `/api/items/{id}` calls, where `{id}` is some key, e.g. `/api/items/42`.
- ② This method returns an `Optional<Item>` because it will only find, at most, a single `Item` object. The `{id}` piece of the URI is extracted into an argument using the `@PathVariable` annotation. If the method's variable name matches the template variable, then it will bind directly. If you wish to use a different argument name, you can simply use `@PathVariable("id") String itemId`.
- ③ Again, a simple passthrough to Spring Data's `findById(...)` method does the trick.

To move into the arena of creating new `Item` objects for your inventory, you need to code something a little differently. Because HTTP POST calls are not idempotent, but signal a change in the state of the system instead, it's a perfect match for creating a new entry.

The following web method shows how to create new inventory.  
API for creating new inventory

```
@PostMapping("/api/items") ①
 ResponseEntity<?> addNewItem(@RequestBody Item item) { ②
    Item savedItem = this.repository.save(item);

    return ResponseEntity ③
        .created(URI.create("/api/items/" + //
```

```
        savedItem.getId()))
    .body(savedItem); ④
}
```

- ① The `@PostMapping` annotation indicates this method will respond to POST `/api/items` requests. Since you won't know the new `Item` object's id until it's stored in H2, there's no need to ask for it in the URI.
- ② This method is named `addNewItem`, since it's used to create new inventory. Its input argument has a `@RequestBody` annotation indicating the source of its data will be the request body.
- ③ With the newly saved `Item` in hand, you can now fashion a response message using Spring Web's  `ResponseEntity` helper class. It has methods for `ok`, `created`, `accepted`, `noContent`, `badRequest`, `notFound`, and more. In this case, use `created`. Since it expects a URI, you can fashion a relative one pretty quickly using the object's `id` attribute.
- ④  `ResponseEntity.created` also allows you to return a response body, so you can plug in the `savedItem` object. Spring MVC will then serialize it straight into the response body.

Whew! That's a lot!

In fact, if you didn't catch it all, go and re-read it. Trust me, it pays to understand everything that's happening. It's not complex—it simply packs a punch with what you can capture in a single web method.

And if you can grok what `addNewItem()` is doing, then it's a lot easier to connect with what the following `updateItem` method does.

Replacing an existing piece of inventory

```

@PutMapping("/api/items/{id}") ①
public ResponseEntity<?> updateItem( //
    @RequestBody Item item, ②
    @PathVariable Integer id) { ③

    Item newItem = new Item(id, item.getName(), item.getDescription(),
    ④
        item.getPrice());

    this.repository.save(newItem); ⑤

    return ResponseEntity.created(URI.create("/api/items/" +
id)).build(); ⑥
}

```

- ① `@PutMapping` indicates that this method responds to `PUT /api/items/{id}` calls. `PUT` in RESTspeak means "replace" (or "create" at designated location if it doesn't already exist).
- ② Again, `@RequestBody` signals that the updated `Item` details are in the HTTP request body.
- ③ `PUT` requires an identifier so you use the `{id}` that's provided as a `@PathVariable`.
- ④ It's important recognize that the incoming `Item` object's `id` value probably won't match the `{id}` in the URI. Therefore, you should create a new `Item` with the existing data copied in.
- ⑤ `save()` the new `Item` object.
- ⑥ Create an **HTTP 201 Created** using Spring Web's `ResponseEntity.created()` helper method, providing a URI rolled by hand.

This operation is slightly different than the `POST` operation shown previously, but not by much.

Now with this much of an API put together, would you like to start generating some documentation?

## Creating an API portal

After publishing a web service, you want your users to know how to use it. The best way is to stand up an API portal.

Spring REST Docs has your back. By adding it to your project, it will be a snap to write custom material combined with examples of your API. Built upon the tried-and-true Asciidoc tool, Spring REST Docs makes it super simple to capture the details you need.

To prep your project for writing Asciidoc, add this to `pom.xml`. Configuring Maven to generate snippets of web service interaction

```
<plugin>
  <groupId>org.asciidoctor</groupId>
  <artifactId>asciidoctor-maven-plugin</artifactId>
  <version>1.5.8</version>
  <executions>
    <execution>
      <id>generate-docs</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>process-asciidoc</goal>
      </goals>
      <configuration>
        <backend>html</backend>
        <doctype>book</doctype>
      </configuration>
```

```
</execution>
</executions>
<dependencies>
  <dependency>
    <groupId>org.springframework.restdocs</groupId>
    <artifactId>spring-restdocs-asciidoc</artifactId>
    <version>${spring-restdocs.version}</version>
  </dependency>
</dependencies>
</plugin>
```

asciidoc-maven-plugin provides the means to transform AsciiDoc (.adoc) files into HTML. Spring REST Docs, as you'll soon see, provides a mechanism to automatically generate key bits of AsciiDoc. When this plugin runs, its HTML results will end up in target/generated-docs.

By default, Spring REST Docs expects AsciiDoc material in /src/main/asciidoc, so you need to create src/main/asciidoc/index.adoc. With that file created, you can draft the opening lines of your new API portal.

Writing first bits of material for your API portal

```
= Hacking with Spring Boot API Portal
```

When you create a web service, developers want to see how to use it. Thanks to Spring REST Docs, you can generate all the interactions needed via test cases that in turn generate readable documentation. A match made in heaven, or at least, in your IDE.

By running the following request:

So far, so good. It's not much. The good stuff is coming.



**AsciiDoc** is the standard, and **Asciidoctor** is a Ruby project that implements the AsciiDoc standard. Thanks to its strong community and the magic of JRuby, there are lots of Java utilities including **AsciidoctorJ** as well as support in both Maven and Gradle. In fact, this book was written using Asciidoctor.

So what is needed to write good API docs? Examples. Lots of examples.

You can immediately start writing them by hand in `index.adoc`. But if you didn't get the hint, there is a much better way to connect this to the API you're building.

To get started, you'll need some test utilities.

Adding Spring REST Docs support for `WebTestClient` to your build.

```
<dependency>
  <groupId>org.springframework.restdocs</groupId>
  <artifactId>spring-restdocs-webtestclient</artifactId>
  <scope>test</scope>
</dependency>
```

This test-scoped dependency, `spring-restdocs-webtestclient`, provides extra hooks to Spring Framework's `WebTestClient` toolkit aimed at your Spring MVC controllers.



There are other modules in case you are using Spring WebFlux. BTW, go and grab your copy of [Hacking with Spring Boot 2.3: Reactive Edition](#) where there is a complementary chapter on building reactive APIs!

If your plans are to generate the documentation and release them outside of your application's lifecycle, then you could argue you're done. But something that is awfully tricky to avoid is stale material. How many times have you visited a website only to find it doesn't match the latest and greatest details of the app itself?

Spring Boot and its cool ability to serve up static web content makes it the *perfect* place to not just run your application, but to also host relevant documentation about your application. To do that, you simply need to close the loop and copy what Spring REST Docs will soon produce into the right location.

Copying the generated documentation into the JAR file

```
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.7</version>
  <executions>
    <execution>
      <id>copy-resources</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>copy-resources</goal>
      </goals>
```

```
<configuration>
    <outputDirectory>
        ${project.build.outputDirectory}/static/docs
    </outputDirectory>
    <resources>
        <resource>
            <directory>
                ${project.build.directory}/generated-docs
            </directory>
        </resource>
    </resources>
</configuration>
</execution>
</executions>
</plugin>
```

This plugin will copy everything from target/generated-docs into target/classes/static/docs.

With this in place, you can now start writing test cases that exercise your API and capture the results.

### Configuring a test class for REST Docs

```
@WebMvcTest/controllers = ApiItemController.class) ①
@AutoConfigureRestDocs ②
public class ApiItemControllerDocumentationTest {

    private WebTestClient webTestClient; ③

    @MockBean InventoryService service; ④

    @MockBean ItemRepository repository; ⑤

    @BeforeEach
    void setUp(@Autowired MockMvc mockMvc, @Autowired
```

```

RestDocumentationContextProvider restDocumentation) {
    this.webTestClient = MockMvcWebTestClient //
        .bindTo(mockMvc) //
        .filter(documentationConfiguration(restDocumentation)) //
        .build();
}

// ...

}

```

- ① `@WebMvcTest` signals Spring Boot to autoconfigure enough stuff to test Spring MVC controllers, and no more. And by plugging in `ApiItemController.class`, this test case will only start a single Spring MVC controller, making it nicely-focused.
- ② `@AutoConfigureRestDocs` is another Spring Boot annotation that lets you sidestep configuring Spring REST Docs by hand.
- ③ Since you are using `spring-restdocs-webtestclient`, you can easily create an instance of `WebTestClient`.
- ④ When `ApiItemController` is spun up, `@WebMvcTest` would normally complain about `InventoryService` as a missing bean. To avoid having to set that up, just use Boot's `@MockBean` annotation to inject a mock version of this collaborator.
- ⑤ `ApiItemController` also needs an instance of `ItemRepository` to function, so marking this as a `@MockBean` is also needed to get off the ground.

With these bits in place, you are now ready to write your first *document-generating test case!*

### Writing your first document-generating test case

```

@Test
void findingAllItems() {
    when(repository.findAll()).thenReturn( ①

```

```
Arrays.asList(new Item(1, "Alf alarm clock", //  
    "nothing I really need", 19.99));  
  
this.webTestClient.get().uri("/api/items") //  
    .exchange() //  
    .expectStatus().isOk() //  
    .expectBody() //  
    .consumeWith(document("findAll",  
        preprocessResponse(prettyPrint()))); ②  
}
```

- ① This is textbook Mockito where you configure `repository.findAll()` to return back an Array containing a single `Item`.
- ② Here is where the magic happens. `document(...)` is a Spring REST Docs static function that hooks it into the test machinery. With these settings, all generated snippets will be put under `target/generated-snippets/findAll`.

This test method exercises one piece of the API, the `/api/items` route. It uses mocks to properly create collaborating data.

It verifies the HTTP response is ok. You can assert more, but the goal is capturing the JSON response.

And that's where `document(...)` steps in. In the fragment above, `document(...)` has two parameters: the name of the snippets and `preprocessResponse(prettyPrint())`.

The first argument causes a `findAll` folder to be created with several `.adoc` files. The second argument applies "pretty printing" to the JSON.

Why don't you write one more test case before you assemble the snippets into an API portal?

Writing a document-generating test case for creating new objects

```
@Test
void postNewItem() {
    when(repository.save(any())).thenReturn( //
        new Item(1, "Alf alarm clock", "nothing important", 19.99));

    this.webTestClient.post().uri("/api/items") ①
        .bodyValue(new Item("Alf alarm clock", "nothing important",
19.99)) ②
        .exchange() //
        .expectStatus().isCreated() ③
        .expectBody(Item.class) //
        .consumeWith(document("post-new-item",
preprocessResponse(prettyPrint())))); ④
}
```

- ① For this test case, use WebTestClient's put() operator.
- ② Here is where you pick the content to post to the web method.
- ③ Verify that you're getting back an **HTTP 201 Created** status code.
- ④ All the snippets for this test will be gathered under a post-new-item folder.

Apart from these differences to the previous test method, it's quite similar and will produce its own AsciiDoc snippets you can further add to your documentation portal.

With your test case written, custom reference material drafted, and every one configured, you're ready to engage!

This involves more than just running test case. You need to run Maven's prepare-package phase.  
Generating API portal documents

```
$ ./mvnw clean prepare-package

...
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- asciidoctor-maven-plugin:1.5.3:process-asciidoc (generate-
docs) @ hacking-spring-boot-ch6-classic ---
[INFO] Rendered /Users/gturnquist/personal/hacking-spring-boot-
code/6-classic/src/main/asciidoc/index.adoc
[INFO]
[INFO] --- maven-resources-plugin:2.7:copy-resources (copy-resources)
@ hacking-spring-boot-ch6-classic ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.191 s
[INFO] Finished at: 2021-02-27T09:16:45-05:00
[INFO] -----
```

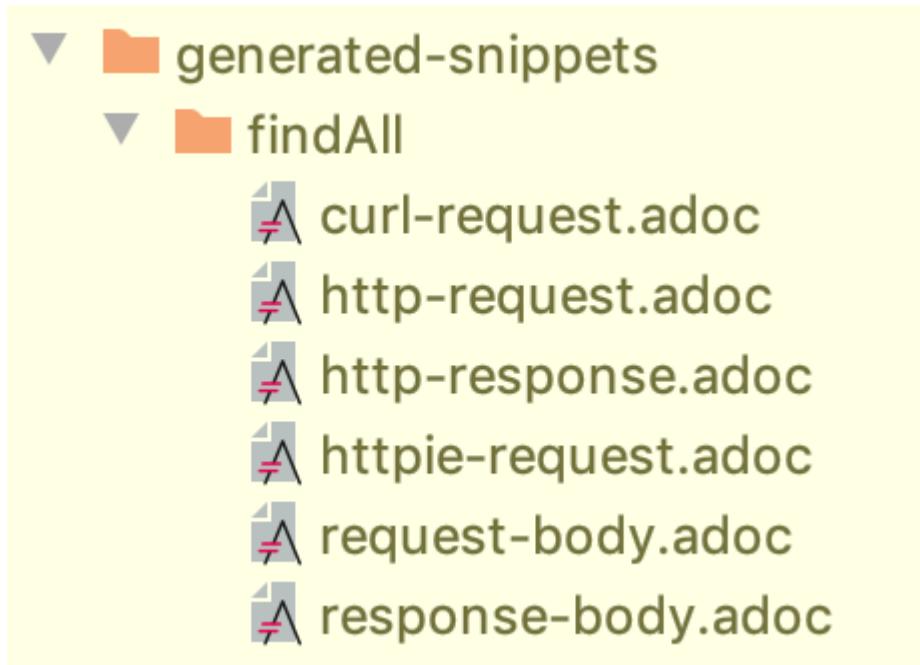
When you run Maven's prepare-package task, several "snippets" are generated including:

- request commands for both cURL and HTTPie
- request and response messages in HTTP format

- request inputs in JSON
- response outputs in JSON

They are gathered in the `/target/generated-snippets` folder, based upon the name you provided to `document()`.

Snippets generated by Spring REST Docs in the `findAll` folder



In the first test case, you can see all the snippets involved in making the request.

The first snippet shows how to execute that request through cURL.

`curl-request.adoc` snippet

```
$ curl 'http://localhost:8080/api/items' -i -X GET
```

There is another snippet showing the actual message transmitted over the wire.

#### http-request.adoc snippet

```
GET /api/items HTTP/1.1
Host: localhost:8080
```

Finally, there's the HTTP response message.

#### HTTP response

```
[ {
  "id" : 1,
  "name" : "Alf alarm clock",
  "description" : "nothing I really need",
  "price" : 19.99
} ]
```

These are quite easy to import into your API documentation, allowing you to tie together automated test cases with reference materials for your community to read.

To leverage all these snippets, simply update your `index.adoc` file.

#### API documentation with snippets

```
= Hacking with Spring Boot API Portal
```

When you stand up a web service, people will want to see how to use it. Thanks to Spring REST Docs, you can generate all the interactions needed via test cases that in turn generate readable documentation. A match made in heaven, or at least, in your IDE.

By running the following request:

```
include::{snippets}/findAll/curl-request.adoc[]
```

...the 'ApiController' will yield the following response:

```
include::{snippets}/findAll/response-body.adoc[]
```

Fan of HTTPie? You can issue the following command:

```
include::{snippets}/findAll/httpie-request.adoc[]
```

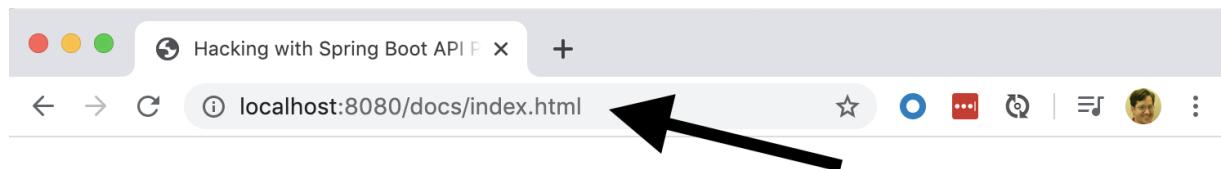
...and get the same response. The choice is yours.



If you check out the [source of this book](#), you'll find the include directives above are prefixed with a backslash. This was so the manuscript, written in AsciiDoc, would render what you see above and NOT insert the snippets. When you create your own API portal, just use a plain old include directive to pull in your own snippets.

You now have everything in place. Just run `./mvnw spring-boot:run` and your API portal is now a part of the application.

## Viewing the API portal



## Hacking with Spring Boot API Portal

When you stand up a web service, people will want to see how to use it. Thanks to Spring REST Docs, you can generate all the interactions needed via test cases that in turn generate readable documentation. A match made in heaven, or at least, in your IDE.

By running the following request:

```
$ curl 'http://localhost:8080/api/items' -i -X GET
```

...the `ApiItemController` will yield the following response:

```
[ {  
    "id" : "item-1",  
    "name" : "Alf alarm clock",  
    "description" : "nothing I really need",  
    "price" : 19.99  
} ]
```

Fan of HTTPie? You can issue the following command:

```
$ http GET 'http://localhost:8080/api/items'
```

- You can see the path to access the API portal.
- The snippets generated by Spring REST Docs are nicely rendered.
- Your custom prose is sprinkled around the autogenerated stuff.

If you check Spring Boot Maven Plugin's code, you'll discover that the `run` goal runs during the `test-compile` phase. That means that it doesn't even get to the `prepare-package` phase typically wired for building the API portal. If you want to both build the API portal *and* run it in one fell swoop, you have to do something like `./mvnw verify spring-boot:run`.

All in all, it provides the perfect balance between written documentation and automated test cases with your API.

Spring REST Docs has all sorts of preprocessors you can apply both to the request (`preprocessRequest`) as well as the response (`preprocessResponse`). These give you the ability to tailor the test results.

*Table 5. Spring REST Docs pre-built preprocessors*

Preprocessor	Description
<code>prettyPrint()</code>	Print the request or response in human-readable format.
<code>removeHeaders(String... headerNames)</code>	Remove any header you don't want. HINT: Spring Framework's <code>HttpHeaders</code> utility class contains all the standard header names as constants. Use them!
<code>removeMatchingHeaders(String... headerNamePatterns)</code>	Remove any headers based on regular expression patterns.

---

Preprocessor	Description
<code>maskLinks()</code>	Replace any <code>href</code> links entries (usually found in hypermedia JSON formats like HAL) with a "..." masked value.
<code>maskLinks(String mask)</code>	Replace any <code>href</code> links entries with your own masking value.
<code>replacePattern(Pattern pattern, String replacement)</code>	Replace any regular expression pattern.
<code>modifyParameters()</code>	Alter the request by adding, setting, and removing parameters through a fluent API.
<code>modifyUris()</code>	Modify URIs found in requests and responses through a fluent API.

---

If you need another option that's not provided, you can write your own `OperationPreprocessor` and register it! But instead of implementing this interface, simply extend `OperationPreprocessorAdapter` and override whatever parts you need.



In a Spring REST Docs test case, remember that the test data in your mocks will appear in your documentation. If you have a certain theme to all your examples, be sure to continue it. (HINT: LoTR deemed cool!)

## The challenges of API evolution

Okay, building a JSON API wasn't that hard. And you saw that standing up an API portal is also pretty simple using Spring REST Docs.

However, one of the biggest costs you'll find isn't implementing APIs or necessarily crafting documentation.

It's API evolution.

A problem developers have faced, over and over, is change. When you need to add something to an API, what do you do? Break existing users? Offer multiple versions? Or let your existing API provide backwards compatibility?

Be forewarned—what you *can* do and what you *should* do can be different.

Jean-Jacques Dubray conducted a study titled "Understanding the Costs of Versioning". Unfortunately, it's no longer available,

but a summary [can be found on InfoQ](#). A major takeaway from that study is carried in this quote:

**“** *The key point that [you need] to understand is that even if the cost to your consumers may look small to you, it is not just a pure cost, it is risks, disrupted project plans, unavailable budgets...with changes that often have no immediate business value to an existing consumer who was not expecting any change to API.*

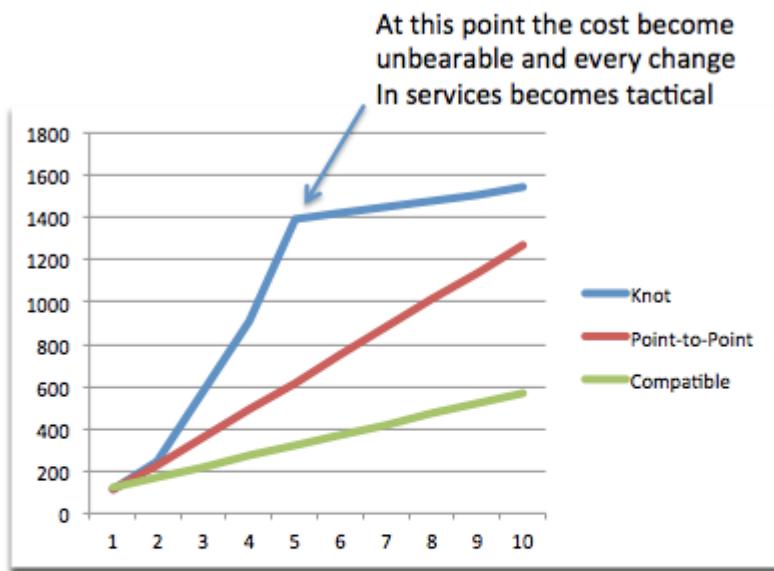
~ Jean-Jacques Dubray *Understanding the Costs of Versioning* ([reprinted on InfoQ](#))

In the study, Dubray describes three forms of making changes to an API:

- **The Knot** - All API consumers are tied to a single version. When the API changes, all consumers must change, creating a great ripple. (top line on the chart)
- **Point-to-Point** - Every server is left running in production and consumers may migrate at their leisure. (middle line on the chart)

- **Compatible Versioning** - All clients talk to the same compatible API. (bottom line on the chart)

Graphed results of Dubray's study



Versions	1	2	3	4	5
Knot	110	250	580	910	1390
Point-to-Point	110	230	360	490	620
Compatible	120	170	220	270	320
Gains (Comp. vs P2P)	-9%	26%	39%	45%	48%

- The x-axis depicts each successive evolution of the API.
- The y-axis depicts subsequence cost to implement.

As shown in the chart above, The Knot had costs that rise the fastest. While there was little to maintain on the server, the impact was drastic. Clients were forced to upgrade, whether they needed the upgraded features or not.

Serving multiple versions (Point-to-Point) significantly brought down the cost, but it rose nonetheless. Though it certainly had

its price. Supporting multiple versions of an API, while lessening the impact on clients, introduces additional costs on the server team.

The final solution, a backward-compatible API had the lowest increase in cost. By supporting existing clients but allowing newer features to roll out *on the same API*, clients are able to migrate when it best suits them. And the server team isn't left holding the bag to sustain multiple versions, some of which may *never* get used.

So how can you implement a backward-compatible service that smooths the way for clients to upgrade? By introducing hypermedia into your API.

## Creating a hypermedia-based web service

Hypermedia is a concept we're all familiar with in one form or another. It was hypermedia that revolutionized the Internet when Tim Berners-Lee created the World Wide Web. Other protocols existed at the time for serving and finding documents (Gopher, Archive, FTP, and others), but the concept of wedging data with navigational links, a.k.a. hypermedia, won out in the end.

Adding hypermedia to APIs and empowering them with the same concepts that made the web so powerful can introduce similar flexibility and ability to evolve.

Hypermedia itself can be quite taxing to code on your own. That's why there's Spring HATEOAS. Combined with it's support for Spring MVC, you can craft hypermedia representations of your services with great agility.

To refresh your memory, check out the web method from earlier in this chapter used to return an Item.

Original method to fetch Item object

```
@GetMapping("/api/items/{id}") ①
Optional<Item> findOne(@PathVariable Integer id) { ②
    return this.repository.findById(id); ③
}
```

That simple Spring MVC method leveraged a Spring Data repository to yield the following JSON structure.

Simple JSON data representation

```
{
    "id" : 1,
    "name" : "Alf alarm clock",
    "description" : "nothing I really need",
    "price" : 19.99
}
```

That's not the only information of value. In that Item JSON, it would be handy to have the details needed for the client to

perform other operations (PUT, PATCH, DELETE, etc.)

To build a richer format, first you must add Spring HATEOAS to your application.

Adding Spring Boot's HATEOAS starter

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

spring-boot-starter-hateoas brings in the necessary bits to activate Spring HATEOAS.

With this added to the build, it's now possible to write a more detailed web controller method.

Defining a hypermedia API

```
@RestController
public class HypermediaItemController {

    private final ItemRepository repository;

    public HypermediaItemController(ItemRepository repository) {
        this.repository = repository;
    }

    // ...

}
```

This controller is exactly like `ApiItemController` when it comes to setup. And that's a good thing! To get started, there's no extra ceremony or incantations to register a controller ready to serve up hypermedia.

To look up a single Item, the code is a bit longer.

### Building hypermedia for a single Item

```
@GetMapping("/hypermedia/items/{id}")
EntityModel<Item> findOne(@PathVariable Integer id) {
    HypermediaItemController controller =
    methodOn(HypermediaItemController.class); ①

    Link selfLink = linkTo(controller.findOne(id)).withSelfRel(); ②

    Link aggregateLink = linkTo(controller.findAll()) //
        .withRel(IanaLinkRelations.ITEM); ③

    return this.repository.findById(id) ④
        .map(item -> EntityModel.of(item, selfLink, aggregateLink)) //
        .orElseThrow(() -> new IllegalStateException("Couldn't find
item " + id));
}
```

- ① Use Spring HATEOAS's `methodOn` operator to get a hold of the controller.
- ② Use the `linkTo` operator to create a link to the controller's `findOne()` method. Because you're linking to the very method you're in, give it a "self" link relation.
- ③ Look up the controller's `findAll()` method to provide a link to the aggregate root. Give it an IANA standard link relation of "item".
- ④ Combine it all into a hypermedia enabled response.

When forming hypermedia representations, the first step is to combine domain objects with links. To ease this task, Spring HATEOAS provides the following vendor-neutral types:

- `RepresentationModel` - base type for defining domain objects that also have links.
- `EntityModel` - extension of `RepresentationModel` that *wraps* a domain object and then provides the means to add links.
- `CollectionModel` - extension of `RepresentationModel` that wraps a Collection of domain objects and allows you to add links.
- `PagedModel` - extension of `CollectionModel` that includes paging metadata.

These four types combined with `Link` and `Links` are the cornerstone of Spring HATEOAS's ability to let you define hypermedia. Return one of these types from a web method, and Spring HATEOAS's custom serializers will kick in and generate hypermedia.

In REST, the "thing" you interact with is called a **resource**. The web methods you write in your Spring MVC controllers are truly the resources being defined. But these HATEOAS types give you the tools to model representations served by those resources.

This web controller can be tested, just like the plain old JSON API was earlier in this chapter, using Spring REST Docs. Start by

creating a new test. And like the previous one, only focus on one controller, the hypermedia one.

### Configuring a test class for hypermedia API

```
@WebMvcTest/controllers = HypermediaItemController.class
@AutoConfigureRestDocs
public class HypermediaItemControllerDocumentationTest {

    private WebTestClient webTestClient;

    @MockBean InventoryService service;

    @MockBean ItemRepository repository;

    @BeforeEach
    void setUp(@Autowired MockMvc mockMvc, @Autowired
    RestDocumentationContextProvider restDocumentation) {
        this.webTestClient = MockMvcWebTestClient //
            .bindTo(mockMvc) //
            .filter(documentationConfiguration(restDocumentation)) //
            .build();
    }

    // ...

}
```

Once again, the setup for a documentation test case is identical. The only difference is pointing it at `HypermediaItemController`. Bottom line—capturing HTTP responses from your APIs is easy, hypermedia or not.

To build things up, it's nice to start with fetching a single Item object. This shows the differences between a bare record (what you built earlier in the chapter) compared to one with hypermedia controls added.

### Testing a hypermedia endpoint

```
@Test
void findOneItem() {
    when(repository.findById(1)).thenReturn(Optional.of( //
        new Item(1, "Alf alarm clock", "nothing I really need",
19.99)));

    this.webTestClient.get().uri("/hypermedia/items/1") //
        .accept(MediaType.APPLICATION_JSON) //
        .exchange() //
        .expectStatus().isOk() //
        .expectBody() //
        .consumeWith(document("findOne-hypermedia",
preprocessResponse(prettyPrint())), //
            links( ①
                linkWithRel("self").description("Canonical link to this
`Item`"), ②
                linkWithRel("item").description("Link back to the
aggregate root"))); ③
}
```

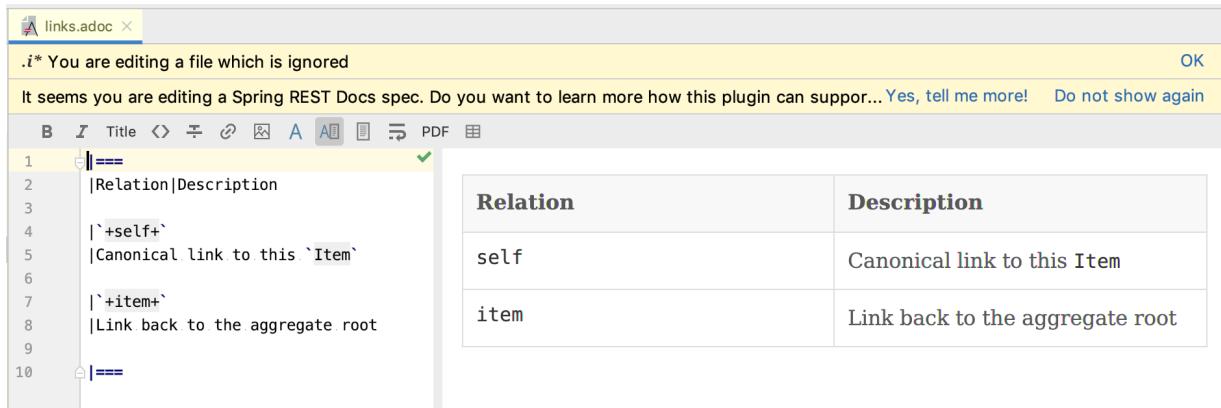
- ① Spring REST Docs has a `links()` function inside its HypermediaDocumentation to gather link descriptors.
- ② `linkWithRel("self")` looks for the `self` link, and documents it as the canonical link using `description()`.
- ③ `linkWithRel("item")` looks for the `item` link, and labels it as the link back to the aggregate root.

Spring REST Docs verifies the link exists, and after doing so, generates a snippet called `links.adoc` (shown below).

The HTTP response body from that test case is shown below.  
Hypermedia JSON data representation

```
{  
    "id" : 1,  
    "name" : "Alf alarm clock",  
    "description" : "nothing I really need",  
    "price" : 19.99,  
    "_links" : {  
        "self" : {  
            "href" : "http://localhost:8080/hypermedia/items/1"  
        },  
        "item" : {  
            "href" : "http://localhost:8080/hypermedia/items"  
        }  
    }  
}
```

In the code above, not only are there snippets for the JSON outputs, there is a separate snippet listing each hypermedia link. This screenshot shows it displayed inside an IDE.  
Hypermedia links gathered in a table



The screenshot shows a software interface for editing AsciiDoc files. The title bar says "links.adoc". A yellow status bar at the top indicates ".i\* You are editing a file which is ignored" and "It seems you are editing a Spring REST Docs spec. Do you want to learn more how this plugin can support... Yes, tell me more! Do not show again". Below the status bar is a toolbar with icons for bold, italic, title, etc. The main area has a code editor with the following content:

```
1 |===
2 |Relation|Description
3
4 |`+self+`
5 |Canonical link to this 'Item'
6
7 |`+item+`
8 |Link back to the aggregate root
9
10|===

```

To the right of the code editor is a preview pane containing a table:

Relation	Description
self	Canonical link to this Item
item	Link back to the aggregate root

You can see the raw AsciiDoc snippet on the left and preview on the right. As a bonus, this IDE even has an alert at the top signaling it recognizes this as Spring REST Docs!

From there, you can import links.adoc into your API portal, rendering this.

### Spring REST Docs accumulation of link details

```
|===
|Relation|Description

|`+self+`
|Canonical link to this 'Item'

|`+item+`
|Link back to the aggregate root

|===

```

This creates the perfect mixture of automated testing, custom descriptions supplied in the test case, and your ability to add any extra details before and after the snippet.

# The value of hypermedia

At this point, you've seen how to create hypermedia, how to test it, and how to display it on an API documentation site. But perhaps a more fundamental question in your mind is, "why do all this?"

And that's a fair question. After all, just because you *can* do something doesn't mean you *should*.

The point of hypermedia is to provide not just data, but also information on how to digest that data. That's why hypermedia documents often come with **profile** links—links that will give you a different JSON format that tells you about the data. These **profile** records can sometimes be fed into JavaScript libraries to render create/update forms.

Case in point—[JSON Schema](#). This metadata format, if served by your API, can be piped into [JSON Editor](#), a JavaScript library that will create a popup HTML form.

Using JSON Schema to dynamically create an HTML FORM





JSON Schema profiles are something offered by Spring Data REST, a user of Spring HATEOAS, not Spring HATEOAS itself. There are efforts to migrate this mediatype from Spring Data REST into Spring HATEOAS so everyone can use it.

Another form of metadata supported out of the box by Spring HATEOAS is [ALPS](#) (Application-Level Profile Semantics).

You can build your own profile by writing a supporting web method.

Creating a profile link with metadata

```
@GetMapping(value = "/hypermedia/items/profile", produces =
MediaTypes.ALPS_JSON_VALUE)
public Alps profile() {
    return alps() //
        .descriptor(Collections.singletonList(descriptor() //
            .id(Item.class.getSimpleName() + "-repr") //
            .descriptor(Arrays.stream( //
                Item.class.getDeclaredFields()) //
                .map(field -> descriptor() //
                    .name(field.getName()) //
                    .type(Type.SEMANTIC) //
                    .build()) //
                .collect(Collectors.toList()))) //
            .build())) //
        .build();
}
```

It's left as an exercise for the reader to craft a test case using Spring REST Docs that exercises this method and produces a

snippet of ALPS JSON metadata resembling the following.

Item object represented using ALPS metadata

```
{  
  "version" : "1.0",  
  "descriptor" : [ {  
    "id" : "Item-repr",  
    "descriptor" : [ {  
      "name" : "id",  
      "type" : "SEMANTIC"  
    }, {  
      "name" : "name",  
      "type" : "SEMANTIC"  
    }, {  
      "name" : "description",  
      "type" : "SEMANTIC"  
    }, {  
      "name" : "price",  
      "type" : "SEMANTIC"  
    } ]  
  } ]  
}
```

It's important to understand that dynamic HTML forms aren't the only reason to use hypermedia.

There is a deeper, more fundamental concept at play—connascence.

Connascence is a software quality metric invented by Meiller-Page Jones to describe complexity in software systems. To quote Wikipedia, "In software engineering, two components are connascent if a change in one would require the other to be

modified in order to maintain the overall correctness of the system."<sup>[4]</sup>

When one team owns the entire application including the frontend and the backend, then you already are in a tightly-coupled scenario. Implementing hypermedia would appear to buy you very little, since any changes to the frontend and backend would be conducted by the same team.

However if you have built an API that is used by several outside teams, some internal and some external, then the situation is quite different. Imagine building an order service API that includes information about the state of the order in various fields. Perhaps when the "state" of the order is "PENDING", you can cancel the order. But when it reaches "FULFILLED" you cannot. If clients write logic to decide when to display the "Cancel" button based on the order's "state", they are tightly coupling themselves to the backend.

The backend may add new states. And if clients have built their logic on top of payload data, it will break them. What if you didn't add anything to the flow of carrying out an order, but added internationalization instead? Again, that would cause a massive upset to most clients.

But what if the clients learned how to learn to read links in hypermedia? Then, instead of "guessing" based on payload data, they could instead simply look for a "cancel" link. See it? Then

show it. That would elegantly decouple them from having to know how the ordering process works. All they need to do is show the link when it's offered, putting the backend back in control.

This whole concept of clients trading in domain knowledge and instead embracing protocol knowledge, i.e. swapping order awareness for the ability to read links, the clients could avoid a whole class of potential problems.

This is what REST is all about. Despite what people say (and write on blog sites), REST is:

- Not about pretty URIs, e.g. /orders/23
- Not about Spring MVC + Jackson (or any other technology stack)
- Not about CRUD-over-HTTP as you POST JSON documents to insert new rows into a database.

Instead, REST is about using the same tactics that made the web emerge as the center of commerce, government, and so many other aspects that connect our lives. The pure fact that a web site update doesn't require a web browser update shows that it's possible to upgrade servers without requiring client updates.

If you apply the same concepts and constraints to APIs, as proposed in Dr. Roy Fielding's dissertation, you can also have a

backward-compatible API that will reduce the total cost of maintenance over time.



Many of these details and examples are borrowed from Oliver Drotbohm's [REST Beyond the Obvious](#) talk. See his [video presentation on the same topic](#).

## Adding affordances to your API

**“** When I say Hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the **affordance** through which the user obtains choices and selects actions.

~ Roy Fielding [A Little REST and Relaxation](#) (slide 50) @ ApacheCon 2008

You've created hypermedia and registered some relevant links. But do you feel like something is missing?

So far, the main JSON format shown has been either plain old JSON or the link-loaded hypermedia format of HAL. It is one of the most widely adopted formats for hypermedia, probably due to its simplicity. And it's the default format Spring HATEOAS will render.

But there's an issue.

If you provide both a GET and a PUT operation at the same URI, a HAL document will render that as one link. Your users will have no idea there are actually two different things available.

Even if you attempt to force a solution by registering different operations using different link relations, it still won't show what attributes the PUT operation needs to effect change.

Leaving it up to the consumer to stumble around doesn't sound very effective. This is where Spring HATEOAS steps in by providing an API to augment your hypermedia with **affordances**.

A classic example is when viewing a single Item resource, you are *afforded* the means to update that resource.

Spring HATEOAS provides the means to connect related methods. In this case, the link to the GET operation can be connected to the PUT operation.

How does this help? It's true that rendered as HAL, you'll still see a single link. But another hypermedia format like HAL-FORMS can take advantage of this connection to render additional information. Any affordance-friendly mediatype can leverage this extra metadata.

To compare plain hypermedia with affordance hypermedia, the usage of Spring HATEOAS's Affordances API will be done in a *separate* controller, `AffordancesItemController`.

Before creating this connection between GET and PUT, you must first define the PUT.

### Updating a single Item resource

```
@PutMapping("/affordances/items/{id}") ①
public ResponseEntity<?> updateItem(@RequestBody EntityModel<Item>
itemEntity, ②
    @PathVariable Integer id) {
    Item content = itemEntity.getContent();
    Item newItem = new Item(id, content.getName(), ③
        content.getDescription(), content.getPrice());

    this.repository.save(newItem); ④

    return ResponseEntity.noContent() //
        .location(findOne(id).getRequiredLink(IanaLinkRelations.SELF).toUri()
    ) ⑤
        .build();
}
```

- ① `@PutMapping` indicates that this web method will respond to HTTP PUT calls, the REST verb for updating existing resources.
- ② This method has a `@RequestBody` input parameter. In this case, it's wrapped as a `EntityModel<Item>`. `EntityModel` indicates that the client could send in either a bare `Item` object, or it could be a hypermedia variant. Either way, this method will handle it.

- ③ After extracting the content, create a new Item object, plugging in the provided {id}. This ensures a complete overwrite of any existing H2 record at the same id, or a new one if none exists.
- ④ Save the new Item object.
- ⑤ Extract the **self** link and convert it to a URI. Then use Spring Framework's ResponseEntity helper methods to fashion an HTTP **204 No Content** response with its Location header populated with the URI.

To simplify, ingest the Item data, store it under the id location, and then reuse `findOne(id)` to extract a URI to send back to the consumer.

`findOne()` is not defined. At least not in this affordance-based controller. But with `updateItem()` in place, you can borrow what you coded in `HypermediaItemController` and create it.

Linking two web methods through an affordance

```

@GetMapping("/affordances/items/{id}") ①
EntityModel<Item> findOne(@PathVariable Integer id) {
    AffordancesItemController controller =
    methodOn(AffordancesItemController.class); ②

    Link selfLink = linkTo(controller.findOne(id)) //
        .withSelfRel() //
        .andAffordance(afford(controller.updateItem(null, id))); ③

    Link aggregateLink = linkTo(controller.findAll()) //
        .withRel(IanaLinkRelations.ITEM);

    return this.repository.findById(id) //
        .map(item -> EntityModel.of(item, selfLink, aggregateLink)) //
        .orElseThrow(() -> new IllegalStateException("Couldn't find

```

```
    item " + id));
}
```

- ① This controller has a prefix of /affordances to illustrate the differences with the previous controller.
- ② Get a hold of the `AffordancesItemController`.
- ③ This is just like the earlier version of `findOne()`, except it uses the `andAffordance()` operator. That operator connects the `updateItem()` method back to this method's `self` link.

You can actually link more than one related method. For example, if you *also* had a method that responded to `DELETE /affordances/items/{id}`, you could include it as well.



When it comes to building affordances, it isn't critical to provide actual data in the inputs. However, when possible, such as using the `id` field, go ahead and provide it. Sometimes those methods need path variable data to build sublinks.

There are different ways to inspect your handiwork. You could run the service or craft a unit test. But why not use it as a chance to capture AsciiDoc snippets to put into your documentation?

Given you just launched another Spring MVC controller to show the differences in adding affordances, go ahead and create `AffordancesItemControllerDocumentationTest`.

Create an affordance-based documentation test class

```

@WebMvcTest/controllers = AffordancesItemController.class) ①
@AutoConfigureRestDocs ②
public class AffordancesItemControllerDocumentationTest {

    private WebTestClient webTestClient; ③

    @MockBean InventoryService service; ④

    @MockBean ItemRepository repository; ⑤

    @BeforeEach
    void setUp(@Autowired MockMvc mockMvc, @Autowired
    RestDocumentationContextProvider restDocumentation) {
        this.webTestClient = MockMvcWebTestClient //
            .bindTo(mockMvc) //
            .filter(documentationConfiguration(restDocumentation)) //
            .build();
    }

}

```

This test case is pointed at that new controller you just built. And as a first test case, why not exercise `findOne()` through a `GET /affordances/items/{id}` call?

### Testing affordances with Spring REST Docs

```

@Test
void findSingleItemAffordances() {
    when(repository.findById(1)).thenReturn(Optional.of( //
        new Item(1, "Alf alarm clock", "nothing I really need",
        19.99)));

    this.webTestClient.get().uri("/affordances/items/1") ①
        .accept(MediaType.APPLICATION_JSON) ②
        .exchange() //

```

```
.expectStatus().isOk() //
.expectBody() //
.consumeWith(document("single-item-affordances", //
    preprocessResponse(prettyPrint())))); ③
}
```

- ① Point WebTestClient at the alternate URI (again, use the "real" one for your stuff).
- ② Configure the Accept header to request a HAL-FORMS response. Otherwise, you'll just get a HAL document that won't look any different!
- ③ Write this alternative response in a single-item-affordances set of AsciiDoc snippets.

Run `./mvnw prepare-package`, and you'll have a new chunk of JSON to examine and potentially include in your API portal!

### HAL-FORMS response body of `findOne()`

```
{
  "id" : 1,
  "name" : "Alf alarm clock",
  "description" : "nothing I really need",
  "price" : 19.99,
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/affordances/items/1"
    },
    "item" : {
      "href" : "http://localhost:8080/affordances/items"
    }
  },
  "_templates" : {
    "default" : {
      "method" : "put",
      "properties" : [ {
```

```
        "name" : "description"
    }, {
        "name" : "id"
    }, {
        "name" : "name"
    }, {
        "name" : "price"
    } ]
}
}
```

Ahh. There it is—a HAL-FORMS document. A rendering of data combined with navigational links *as well as* the information needed to perform a PUT down in the **\_templates** section.

The only thing needed to round out your API is creating a link at the aggregate root and its affordances.

### Generating a list of Item objects with affordances

```
@GetMapping("/affordances/items")
CollectionModel<EntityModel<Item>> findAll() {
    AffordancesItemController controller =
methodOn(AffordancesItemController.class);

    Link aggregateRoot = linkTo(controller.findAll()) //
        .withSelfRel() //
        .andAffordance(afford(controller.addNewItem(null))); ①

    List<EntityModel<Item>> entityModels = //
        StreamSupport.stream(this.repository.findAll().spliterator(),
false) ②
            .map(item -> findOne(item.getId())) ③
            .collect(Collectors.toList());
```

```
    return CollectionModel.of(entityModels, aggregateRoot); ④
}
```

- ① Use the `andAffordance()` operator to connect the `self` link to (not yet created) `addNewItem()`.
- ② Fetch all `Item` objects from H2.
- ③ Map them into the `findOne()` method.
- ④ Using the list of `Item` objects, combine it with the link to the aggregate root to form a `CollectionModel`.



The aggregate root link is combined with the `list` of `Item` objects, not each `Item` by itself.

Adding a new `Item` to inventory isn't hard to noodle out.

### Adding a new item

```
@PostMapping("/affordances/items") ①
ResponseEntity<?> addNewItem(@RequestBody EntityModel<Item>
itemEntity) { ②
    Item content = itemEntity.getContent(); ③
    Item savedItem = this.repository.save(content); ④
    EntityModel<Item> newModel = findOne(savedItem.getId()); ⑤

    return ResponseEntity ⑥
        .created(newModel.getRequiredLink(IanaLinkRelations.SELF).toUri()) //
        .body(newModel.getContent()); //
}
```

- ① `@PostMapping` maps this method onto POST calls.

- ② The incoming data can come with or without links. This is very similar to the PUT method coded earlier.
- ③ Extract the content.
- ④ Save it to H2.
- ⑤ By using `findOne()`, you are handed a `EntityModel<Item>`, links and all.
- ⑥ Fashion an HTTP response using Spring Web's `ResponseEntity.created()` helper. Supply both a `Location` header as well as a response body containing the content.

It's left as an exercise to craft test cases to exercise as well as capture AsciiDoc snippets from the aggregate root.



It is only for demonstration purposes that there are two controllers. You should *really* leverage everything has Spring HATEOAS to offer in a single set of controllers.

There is a lot more than can be done with hypermedia and affordances. What other ways can you imagine linking together web controller methods to build a rich, vibrant API?

## Summary

In this chapter you:

- Created an API to remotely access and alter the system.
- Created test cases using Spring REST Docs to construct a portal documentation site.

- Created a controller that served HAL-based hypermedia with related links.
- Augmented the documentation test cases to capture links and other relevant details for your portal site.
- Introduced affordances and served up HAL-FORMS with templates.
- Captured results in AsciiDoc snippets for your API portal.

In the next chapter, *Messaging with Spring Boot*, you will learn how to use asynchronous message solutions when the time is right.

[4 https://en.wikipedia.org/wiki/Connascence](https://en.wikipedia.org/wiki/Connascence)



# MESSAGING WITH SPRING BOOT

---

We had a hackathon this week, in less than 12 hours we wrote 2 nodejs frontends, 3 microservices in spring boot&data JPA, messaging via RabbitMQ & spring cloud stream, MySQL on amazon RDS and an android app. All code pushed to “production” on PWS. We live in the future.

~ odedia @odedia

In the previous chapter, you delved into creating APIs using Spring Boot. This included classic JSON endpoints that accept inputs and return raw data. It also covered building flexible, backward-compatible APIs using the power of hypermedia a la Spring HATEOAS.

Now it’s time to take your architecture to the next level and decouple certain components using message connections. In

other words, asynchronous messaging can be the means to link different components, whether they run inside the same application or lie in different microservices spread across the network.

In this chapter you'll:

- Learn about the various types of messaging solutions supported by Spring Boot
- Get a glimpse of the various messaging solutions supported by the Spring portfolio, even if not supported directly by Spring Boot
- Delve into AMQP and see how to decouple your frontend web layer and the backend using asynchronous tactics of Spring AMQP.

## Picking your favorite solution

There are dozens of messaging solutions. JMS (Java Messaging Service), Apache Kafka, AMQP (Advanced Message Queuing Technology), Redis, GemFire, Apache Geode, and more. The list seems endless.

They all have many similarities, but optimize for different aspects. This book isn't going to tell which one fits your scenario, because that would depend on the scenario!

But you will get your hands on some ways to do messaging and integrate it properly with asynchronous programming.

## Tackling a problem with a familiar pattern

Before going into specific solutions, it's important to recognize a key trait of the Spring portfolio. And that's **reducing Java complexity**. In fact, that very expression was Rod Johnson's message at the Spring Experience conference in 2008.

First and foremost Spring is designed to simplify the way you build apps. And one of the most powerful mechanisms available is the **template pattern**. So powerful, this pattern is even included in the Gang of Four's famous *Design Patterns* book.

In essence, a **template** is meant to reduce all the complexity of a particular API into its most simple operations. My favorite example is the `JdbcTemplate`. JDBC has a long history of people writing queries, managing connections, and after writing two hundred queries, having to debug the fact that *one* of them forgot to close the result set after the fact!

Spring Framework introduced the `JdbcTemplate`, which required just a few operations to capture the concept of queries and updates. The user only had to provide the SQL, as well as how

to handle the result. Spring handled getting a connection, opening a cursor, forming a result set, and finally closing the result set.

In essence, `JdbcTemplate` took over **resource management**, allowing developers to focus on customer needs.

This pattern was so powerful that it was carried into many other areas: `MailSender`, `JndiTemplate`, `HibernateTemplate`, `JdoTemplate`, and many others.

And this has extended into the many areas of asynchronous messaging. For example:

- **JMS** - the Java standard messaging API - Spring Framework has `JmsTemplate` and `DefaultMessageListenerContainer` to simplify sending messages and consuming messages using a JMS broker.
- **Apache Kafka** - fast-growing broker - Spring for Apache Kafka has `KafkaTemplate` and `KafkaMessageListenerContainer` to simplify sending and consuming messages using Apache Kafka.
- **RabbitMQ** - high-throughput, resilient message broker - Spring AMQP has `AmqpTemplate` and `SimpleMessageListenerContainer` to simplify sending and consuming messages using RabbitMQ.

- **Redis** - fast, popular broker - Spring Data Redis has `RedisTemplate` and `RedisMessageListenerContainer` to ease sending and consuming messages through Redis.

Each of these pub-sub utilities simplify what can be a complex API, and make it very easy to publish messages. These classes also make it easy to register listeners to respond to messages, and then act on them.

In this chapter, you'll delve deeper into using one—RabbitMQ—using the power of Spring AMQP.

## Testing with ease

To dip your toes into message actions, why not start out with a test-based approach by talking to a *real* AMQP broker - **RabbitMQ**.

In an earlier technical tome, I included several pages on how to download, install, and fire up RabbitMQ. That had some undesired side effects such as leftover configurations, so why not nip that in the bud and go for a completely *different* experience?

I'm talking about [Testcontainers](#), the Java testing library that leverages Docker. Testcontainers will fire up just about any database, message broker, or other 3rd-party system you can

imagine (as long as it runs with Docker), and integrate it nicely with your test cases.

And when your test case shuts down, the service shuts down, without leaving behind any state. This makes it super easy to fire up RabbitMQ in a clean state *every single time*.



Testcontainers isn't free. The code is open source, but there is a different cost. You must have Docker installed on your machine for it to work. Anyone else that plans to run your code must *also* have Docker installed. Docker is known for eating up disk space, so you must periodically run `docker system prune` to clean out old containers. But since Docker has become quite prevalent, at least on developer workstations, this isn't the high bar it once was. And given this tactic can be used for all kinds of systems, it's a valuable investment.

To get started, you first must bring in Testcontainers BOM (Bill of Materials). This is an artifact that brings version numbers for various modules.

Setting the Testcontainers version once

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.testcontainers</groupId>
      <artifactId>testcontainers-bom</artifactId>
      <version>1.15.2</version>
      <type>pom</type>
```

```
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

This brings in the Testcontainers BOM, which is a way of setting the version for all Testcontainers modules in one place. It ensures you pick modules that work together.

With that in place, you can now add the test-scoped dependencies needed to focus on RabbitMQ.

A new way of testing

```
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>rabbitmq</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>junit-jupiter</artifactId>
  <scope>test</scope>
</dependency>
```

Testcontainers' `rabbitmq` module will bring in Testcontainers core dependencies that manage Docker along with the module that activates RabbitMQ. In its current version, it will grab the `rabbitmq:3.7.25-management-alpine` image from Docker Hub and spin up an instance for your test cases.

Testcontainers is currently built around JUnit 4. The second dependency is an extension that makes it almost effortless to integrate with JUnit 5, the standard version of JUnit used by Spring Boot 2.4.

When the test cases shut down, the container itself will shut down. With little effort on your end.



[Alpine Linux](#) is a security-oriented, lightweight Linux distribution based on musl libc and busybox.

## Coding a solution

With everything in place, it's time to noodle out *what* your system will do. You're talking about an asynchronous message solution. A simple thing to implement would be receiving a request for a new `Item` object at a web controller and forwarding the information as a message through RabbitMQ. Somewhere else would be a service, listening for messages. That service could then write the new `Item` to H2 using JPA.

This is a very simple concept you can use over and over. And you can adapt it. It's easy to swap out the web controller with something else. Perhaps a message also transmitted via RabbitMQ. Or someone directly invoking the API.

But back to the initial problem: web controller turns synchronous web request into asynchronous message. And this time, you're shooting for a test-first approach.

Spring Boot Test, JUnit 5, and Testcontainers puts this at your fingertips.

Beginning with a test

```
@SpringBootTest ①
@Testcontainers ②
@AutoConfigureMockMvc ③
public class RabbitTest {

    @Container static RabbitMQContainer container = new
    RabbitMQContainer(
        DockerImageName.parse("rabbitmq").withTag("3.7.25-management-
alpine")); ④

    WebTestClient webTestClient;

    @Autowired ItemRepository repository; ⑤

    @DynamicPropertySource ⑥
    static void configure(DynamicPropertyRegistry registry) {
        registry.add("spring.rabbitmq.host",
        container::getContainerIpAddress);
        registry.add("spring.rabbitmq.port", container::getAmqpPort);
    }

    @BeforeEach
    void setUp(@Autowired MockMvc mockMvc) { ⑦
        this.webTestClient = MockMvcWebTestClient //
            .bindTo(mockMvc) //
            .build();
    }
}
```

```
 }
```

There's a lot here, so let's unpack it bit by bit:

- ① `@SpringBootTest` is Boot's annotation to enable all kinds of stuff including autoconfiguration, property support, and possibly an embedded container.
- ② `@Testcontainers` is a JUnit 5 annotation that hooks your test cases into Testcontainers, allowing it to operate.
- ③ `@AutoConfigureMockMvc` is a Spring Boot annotation that autoconfigures Spring Test's MockMvc.
- ④ Create a Testcontainer `RabbitMQContainer`, which will properly manage an instance of RabbitMQ for your test.
- ⑤ Inject a copy of the `ItemRepository` for verification of results.
- ⑥ `@DynamicPropertySource` dynamically adds properties to the Environment using Java 8 Supplier calls. This uses `container::getContainerIpAddress` and `container::getAmqpPort` method handles to grab the hostname and port *after* Testcontainers has spun up the RabbitMQ broker. They are then assigned to the Spring Boot properties used to autoconfigure Spring AMQP.
- ⑦ Grab the autowired `MockMvc` object and use it to create a `WebTestClient`.

While the Spring team generally recommends using **constructor injection** for production components, it's okay to use **field injection** for test classes, given they operate on a different lifecycle.

## Crafting a test case

Before you start drafting that web controller, you need to lay out *how* the controller will handle things.

1. Accept an HTTP POST with the payload to create a new Item object.
2. Turn that payload into a suitable message.
3. Publish it to the broker.

On the receiving end, you'll need to:

1. Listen for new messages.
2. Pull one down.
3. Save it in H2.

Now, craft a test case (and expect it to fail if run right away).

Testing AMQP messaging

```
@Test  
void verifyMessagingThroughAmqp() throws InterruptedException {  
    this.webTestClient.post().uri("/items") ①
```

```

        .bodyValue(new Item("Alf alarm clock", "nothing important",
19.99) //
        .exchange() //
        .expectStatus().isCreated() //
        .expectBody();

Thread.sleep(1500L); ②

this.webTestClient.post().uri("/items") ③
        .bodyValue(new Item("Smurf TV tray", "nothing important",
29.99) //
        .exchange() //
        .expectStatus().isCreated() //
        .expectBody();

Thread.sleep(2000L); ④

Iterable<Item> items = this.repository.findAll(); ⑤

assertThat(items).flatExtracting(Item::getName) //
        .containsExactly("Alf alarm clock", "Smurf TV tray");
assertThat(items).flatExtracting(Item::getDescription) //
        .containsExactly("nothing important", "nothing important");
assertThat(items).flatExtracting(Item::getPrice) //
        .containsExactly(19.99, 29.99);
}

```

- ① Post a new Item object to /items. Verify that an **HTTP 201 Created** status code is returned along with a response body.
- ② To control the ordering of messages, pause 1500 ms, allowing the first message to make it all the way.
- ③ Post a second Item object, verifying the same status.
- ④ Give the broker 2000 ms to finish.
- ⑤ Use ItemRepository to query H2 and verify that two new Item objects have been created that match the inputs.

Right off the bat, it's important to point out that you're verifying against a *real* RabbitMQ broker. Of course, if you run this test case right now, it will fail because there is no code to process things. But that is about to change.

Remember earlier discussing the history of Spring and its mission to reduce Java complexity? Spring AMQP is dedicated to applying the "Spring way" to AMQP, a popular messaging protocol.

The first step is to add Spring AMQP to your project's build file.  
Adding Spring AMQP to the project

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

Next, you need to create a Spring MVC REST controller class to respond to those POST calls.

Configuring a web controller for AMQP messaging

```
@RestController ①
public class SpringAmqpItemController {

    private static final Logger log = //
        LoggerFactory.getLogger(SpringAmqpItemController.class);

    private final AmqpTemplate template; ②

    public SpringAmqpItemController(AmqpTemplate template) {
        this.template = template;
```

```
    }  
}
```

- ① As seen in previous chapters, `@RestController` signals that this class will focus on consuming and rendering JSON payloads, instead of rendering templates.
- ② `spring-boot-starter-amqp` brings Spring AMQP onto the classpath. With that, Spring Boot will autoconfigure an `AmqpTemplate` (which happens to be a `RabbitTemplate` implementation). In this chunk of code, you're merely asking for a copy of that bean to get injected into the controller that needs to publish messages.

If you'll remember from the test code above, you want the controller to respond to HTTP POST `/items` calls, expecting JSON payloads of `Item` data.

To do so, you need to craft a web handler method.  
**Sending AMQP messages from a web controller**

```
@PostMapping("/items") ①  
ResponseEntity<?> addNewItemUsingSpringAmqp(@RequestBody Item item) {  
    ②  
        this.template.convertAndSend( ③  
            "hacking-spring-boot", "new-items-spring-amqp", item);  
        return ResponseEntity.created(URI.create("/items")).build(); ④  
    }  
}
```

- ① This handler responds to POST `/items`.
- ② The `@RequestBody` annotation tells Spring MVC to extract the payload from the body of the request, but only when subscription happens.
- ③ Invoke the `AmqpTemplate` to send the unwrapped `Item` object to the `hacking-spring-boot` exchange with a routing key of `new-items-spring-amqp`.

- ④ Return to the original caller an **HTTP 201 Created** status code with a relative URI back to this endpoint.

## Coding a consumer

With the producer nicely defined in a Spring MVC controller, it's time to switch gears and focus on writing a RabbitMQ consumer. Spring AMQP actually has multiple options. You can use `AmqpTemplate.receive(queueName)`. That's the absolute simplest approach available. And probably not the best solution, especially for high-volume environments.

There are a collection of other methods for either polling for more messages, or for registering a callback. The most flexible and convenient is the `@RabbitListener` approach.

### Consuming AMQP messages

```
@Service ①
public class SpringAmqpItemService {

    private static final Logger log = // 
        LoggerFactory.getLogger(SpringAmqpItemService.class);

    private final ItemRepository repository; ②

    public SpringAmqpItemService(ItemRepository repository) {
        this.repository = repository;
    }

}
```

- ① `@Service` ensures an instance of this class will be created when launched.

- ② The ItemRepository is injected via **constructor injection**.

There is also a Logger, which you'll soon put to good use. But first, it's time to configure what is probably the most convenient way to listen for RabbitMQ messages using Spring AMQP.

### Registering a RabbitMQ message listener

```
@RabbitListener( ①
    bindings = @QueueBinding( ②
        value = @Queue, ③
        exchange = @Exchange(
            value = "hacking-spring-boot", //
            type = ExchangeTypes.TOPIC), ④
        key = "new-items-spring-amqp")) ⑤
public void processNewItemsViaSpringAmqp(Item item) { ⑥
    log.debug("Consuming => " + item);
    this.repository.save(item); ⑦
}
```

- ① @RabbitListener informs Spring AMQP to configure a message listener, ready to consume messages.
- ② @QueueBinding shows how to bind a queue to an exchange.
- ③ @Queue by itself will create a random, non-durable, anonymous queue. If you wish to bind a specific queue, you can supply its name. Additionally, you can declare characteristics like **durable**, **exclusive**, and **autoDelete**,
- ④ @Exchange defines which exchange this queue connects to. In this case, the **hacking-spring-boot** exchange. You can also specify other properties of the desired exchange.
- ⑤ The key is a way for this consumer to declare its preferences. NOTE: If you want to use \* wildcards, then the exchange must be type TOPIC.
- ⑥ This is where you finally get to the meat of the operation. Spring AMQP, upon receiving a Message, will extract the POJO from the payload and

invoke the target method.

- ⑦ The Item object is saved to H2 using a Spring Data repository.

Wow! Didn't expect such a tiny method to have so many callouts. That should illustrate how these annotations pack a real punch.

Spring AMQP has many ways to consume asynchronous messages. The `@RabbitListener` annotation is arguably the most straight-forward way. It supports anonymous queues (what was used in the method above), as well as named queues.



**Anonymous queues vs. named queues**--what's the difference? In an environment where you have different consumers interested in the same message, you must configure things so that they'll get mapped properly.

If two different consumers signed up using the *same* queue, only one of those consumers would get each message. Stated another way—messages on a queue are only consumed by one client. This is handy if you a pool of workers and only one needs to handle a given message.

But if two clients bind to the **same exchange** using the **same routing key** but on different queues, they will *each* get a copy of the message, without the publisher having to be involved. Anonymous queues can guarantee that every instance is listening on their own queue and getting their own copy of each message.

This annotation, when applied to a method, will cause Spring AMQP to register a listener in the background using the most efficient caching and pooling mechanisms available.

Something that isn't quite obvious until you start trying to send messages is that your messages must be **Serializable**. The AMQP spec actually defines how bytes are to be shipped over the wire, not just the APIs. Pretty much any AMQP library easily handles an array of bytes (byte[]). But getting there is up for grabs.

With Spring AMQP, you can choose to implement Java's Serializable interface. It's not difficult, and your messages will be instantly transportable with the code written so far. But it's probably not your best choice.

An alternative is to convert your POJOs into a string representation, like JSON, and then convert that into bytes to ship over the wire. Spring's leverage of Jackson, the JSON serialization library, is a snap if you register the following bean. Configuring JSON-based message serialization

```
@Bean  
Jackson2JsonMessageConverter jackson2JsonMessageConverter() {  
    return new Jackson2JsonMessageConverter();  
}
```

This bean can be placed inside any @Configuration class. This includes the one with @SpringBootApplication annotation, which

itself is a configuration class.

This will automatically activate a Spring Framework `MessageConverter` that translates POJOs to JSON and back, as needed.



How important is it to avoid `java.io.Serializable`? It's no secret that deserialization sidesteps many of the security checks built into Java. This has been the source of previous security attacks, and has been a thorn in the Java team's side. Mark Reinhold, Oracle's chief architect of the Java Platform Group, has referred to `Serializable` as "a horrible mistake in 1997" and has voiced the desire to remove it from the Java specification. It's better to use something like Jackson that doesn't get these "shortcuts" and instead allows stricter control. So in general, I'd recommend using Jackson over `Serializable`, unless there are solid benchmarks showing Jackson causing slowdowns.

With all this in place, it's time to take the JUnit test case, `verifyMessagingThroughAmqp()`, out for a spin.



Want to dial up the "nerd knobs" and see what's happening between Spring AMQP, RabbitMQ, and even Spring Data? Add these logging levels to `src/main/resources/application.properties`.

```
logging.level.org.springframework.amqp=DEBUG
logging.level.org.springframework.messaging=DEBUG
logging.level.com.gregturnquist.hackingspringboot=DEBUG
logging.level.org.springframework.data=DEBUG
spring.mvc.hiddenmethod.filter.enabled=true
```

This will dump out TONS of stuff to chew through. You can remove it later after you're comfortable everything is running correctly.

## Testing with Spring AMQP and RabbitMQ

```
2021-02-20 12:25:57.781 DEBUG CachingConnectionFactory      :
Creating cached Rabbit Channel from
AMQChannel(amqp://guest@127.0.0.1:55249/,1)

2021-02-20 12:25:58.448 DEBUG RabbitTemplate      : Executing
callback RabbitTemplate$$Lambda$1686/2121538923 on RabbitMQ Channel:
Cached Rabbit Channel: AMQChannel(amqp://guest@127.0.0.1:55249/,2),
conn: Proxy@2245ccaa Shared Rabbit Connection:
SimpleConnection@5c78f456 [delegate=amqp://guest@127.0.0.1:55249/,
localPort= 62918]

2021-02-20 12:25:58.448 DEBUG RabbitTemplate      : Publishing
message [(Body:'{"id":null,"name":"Alf alarm
clock","description":"nothing important","price":19.99}')
MessageProperties [headers=
{__TypeId__=com.gregturnquist.hackingspringboot.classic.Item},
contentType=application/json, contentEncoding=UTF-8,
contentLength=84, deliveryMode=PERSISTENT, priority=0,
deliveryTag=0]] on exchange [hacking-spring-boot], routingKey =
```

```
[new-items-spring-amqp]
```

```
2021-02-20 12:25:58.454 DEBUG BlockingQueueConsumer : Storing delivery for consumerTag: 'amq.ctag-g8FR-hJ04FTFZrw-I3BCVg' with deliveryTag: '1' in Consumer@3e8fe7db: tags=[[amq.ctag-g8FR-hJ04FTFZrw-I3BCVg]], channel=Cached Rabbit Channel: AMQChannel(amqp://guest@127.0.0.1:55249/,1), conn: Proxy@2245ccaa Shared Rabbit Connection: SimpleConnection@5c78f456 [delegate=amqp://guest@127.0.0.1:55249/, localPort= 62918], acknowledgeMode=AUTO local queue size=0
```

```
2021-02-20 12:25:58.455 DEBUG BlockingQueueConsumer : Received message: (Body:'{"id":null,"name":"Alf alarm clock","description":"nothing important","price":19.99}' MessageProperties [headers={__TypeId_=com.greglturnquist.hackingspringboot.classic.Item}, contentType=application/json, contentEncoding=UTF-8, contentLength=0, receivedDeliveryMode=PERSISTENT, priority=0, redelivered=false, receivedExchange=hacking-spring-boot, receivedRoutingKey=new-items-spring-amqp, deliveryTag=1, consumerTag=amq.ctag-g8FR-hJ04FTFZrw-I3BCVg, consumerQueue=spring.gen-ejh-27sMTA2sLoE9iXSy1w])
```

```
2021-02-20 12:25:58.463 DEBUG MessagingMessageListenerAdapter : Processing [GenericMessage [payload=Item{id='null', name='Alf alarm clock', description='nothing important', price=19.99}, headers={amqp_receivedDeliveryMode=PERSISTENT, amqp_receivedExchange=hacking-spring-boot, amqp_deliveryTag=1, amqp_consumerQueue=spring.gen-ejh-27sMTA2sLoE9iXSy1w, amqp_redelivered=false, amqp_receivedRoutingKey=new-items-spring-amqp, amqp_contentEncoding=UTF-8, id=a07a7246-932e-95b2-3eef-57c3abf56023, amqp_consumerTag=amq.ctag-g8FR-hJ04FTFZrw-I3BCVg, amqp_lastInBatch=false, contentType=application/json, __TypeId_=com.greglturnquist.hackingspringboot.classic.Item, timestamp=1613845558463}]]
```

```
2021-02-20 12:25:58.464 DEBUG SpringAmqpItemService : Consuming
```

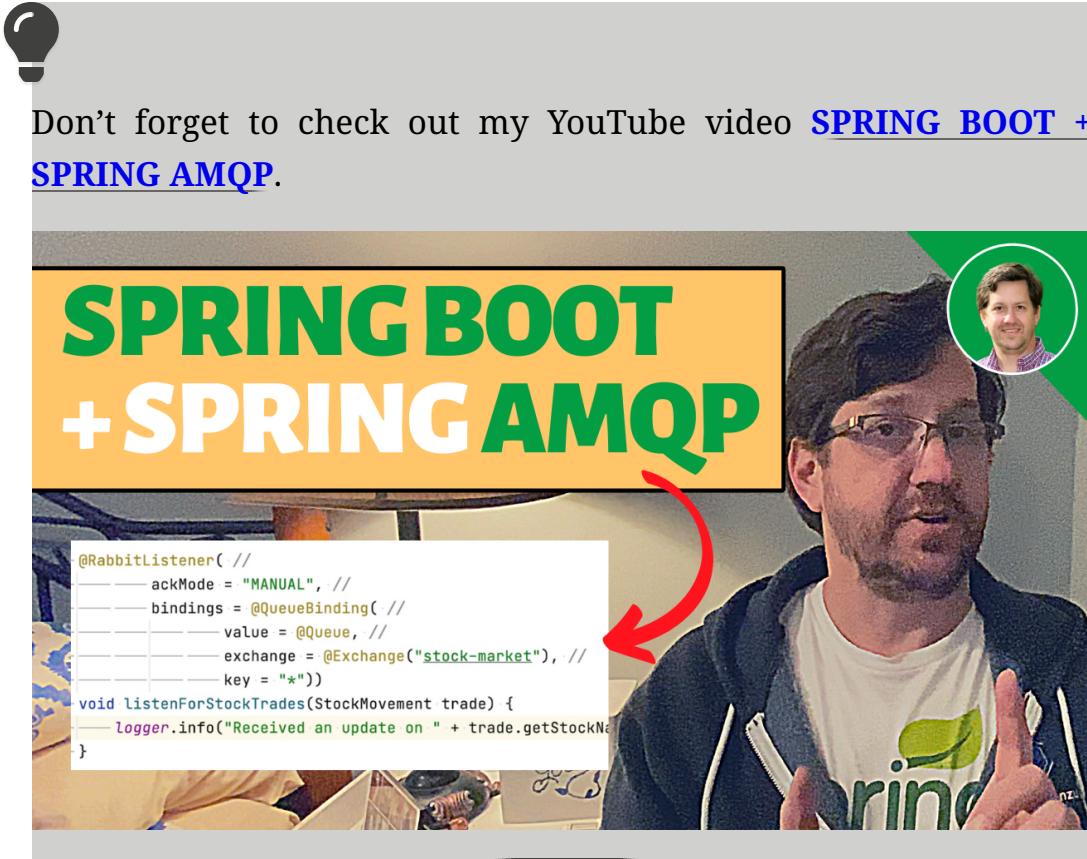
```
=> Item{id='null', name='Alf alarm clock', description='nothing important', price=19.99}
```

This is just a fragment of the console output, and its contents have been trimmed a bit to help with readability. This only shows one of the two messages transported through RabbitMQ.



It still pays to run the test case yourself and comb through every line. This is some powerful stuff to simplify writing a message-based solution.

That's the basics in getting your AMQP messaging solution off the ground in Spring land.



It's simple. It's powerful. It's easy to tailor to your needs.

## Summary

In this chapter you got your hands on RabbitMQ. You used Testcontainers to get started with a messaging solution without having to spend hours or days standing up a broker. Pick a cloud solution like VMware's Tanzu Application Service, and going to production could be just as simple.

But this chapter is aimed at introducing the more general concept of asynchronous message solutions. RabbitMQ isn't the

only option. You can use JMS or something like ActiveMQ, or Kafka, or ... whatever.

The concepts are the same:

- Publishing one message.
- Consuming it with one or more consumers.
- Using the Spring portfolio's various template approaches (`RabbitTemplate`, `RabbitMessageTemplate`, `AmqpTemplate`, `JmsTemplate`, `KafkaTemplate`).

Your knowledge of using Spring AMQP isn't hard to carry over to what may be the most proper messaging solution you pick on your next project.

In this chapter you:

- Set up Testcontainers, RabbitMQ, and Spring AMQP.
- Created a test to verify the expected behavior of the web and the backend.
- Created a Spring MVC controller to accept an incoming, synchronous web call.
- Used `RabbitTemplate` to transmit your message over an asynchronous message broker.
- Configured a RabbitMQ listener using `@RabbitListener` and consumed the generated message.

In the next chapter, *Securing your Application with Spring Boot*, you will learn how to secure your application with Spring Security.



# SECURING YOUR APPLICATION WITH SPRING BOOT

---

It's not real until it's secured.

~ Greg L. Turnquist @gregturn

In the previous chapter, you learned how to connect different components asynchronously using messaging and AMQP.

In this chapter, you will dig into the portion of your application that *must* be a part of it if you wish to go to production: **security**.

I once tinkered with a new web stack. It was pretty nifty. Then I wanted to secure my newly-minted web app. That's when I found out their security solution was still under development. Since I needed an operational demo for an upcoming conference, I set this effort aside and picked something I knew already had security.

An application is no more than a toy until you can lock it down and govern who has access and apply strict authorization controls.

And that's what you'll get to do in this chapter.

In this chapter you'll:

- Configure Spring Security using different user stores
- Apply route-based security settings to HTTP endpoints
- Wire up method-level security to Spring MVC endpoints
- Hook into Spring Security's context to check for operational authorization

## Getting started

To get things off the ground, you can use Spring Boot's absolutely simplest approach: add Spring Security to the project and take off!

Yup, just add Spring Security and then see what Spring Boot does for you.

Adding Spring Security to your Spring Boot project

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

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

In addition to adding Spring Security to your project, this also includes `spring-security-test` in a test-scoped fashion. This will make it super easy to write some security-oriented test cases further down in this chapter.



All the code in this section can be found [here](#).

That's easy enough, right?

If you combined this with the shopping cart application you have been working on throughout this book, what happens if you just run it?

```
.-----.
/\ \ / _ _ ' _ _ _ _ ( _ ) _ _ _ _ \ \ \ \
( ( ) \ _ _ | ' _ | ' _ | ' _ \ \ ' | \ \ \ \
\ \ \ _ _ ) | ( _ ) | | | | | | ( _ | | ) ) ) )
' | _ _ | . _ | _ | _ | _ \ _ , | / / / /
=====|_|=====|_|=====|_|=/_/_/_/
:: Spring Boot ::          (v2.4.4)
```

```
2021-02-23 21:31:06.700  INFO 82765 --- [           main]
c.g.h.c.HackingSpringBootApplication      : Starting
HackingSpringBootApplication using Java 1.8.0_242 on gturnquist-
```

```
a01.vmware.com with PID 82765 (/Users/gturnquist/personal/hacking-with-spring-boot-classic-code/8-classic-quick/target/classes started by gturnquist in /Users/gturnquist/personal/hacking-with-spring-boot-classic-code)
```

```
...more Spring Boot messages...
```

```
Using generated security password: 52855adb-bd8d-4b81-9558-87c3ee28f9d9
```

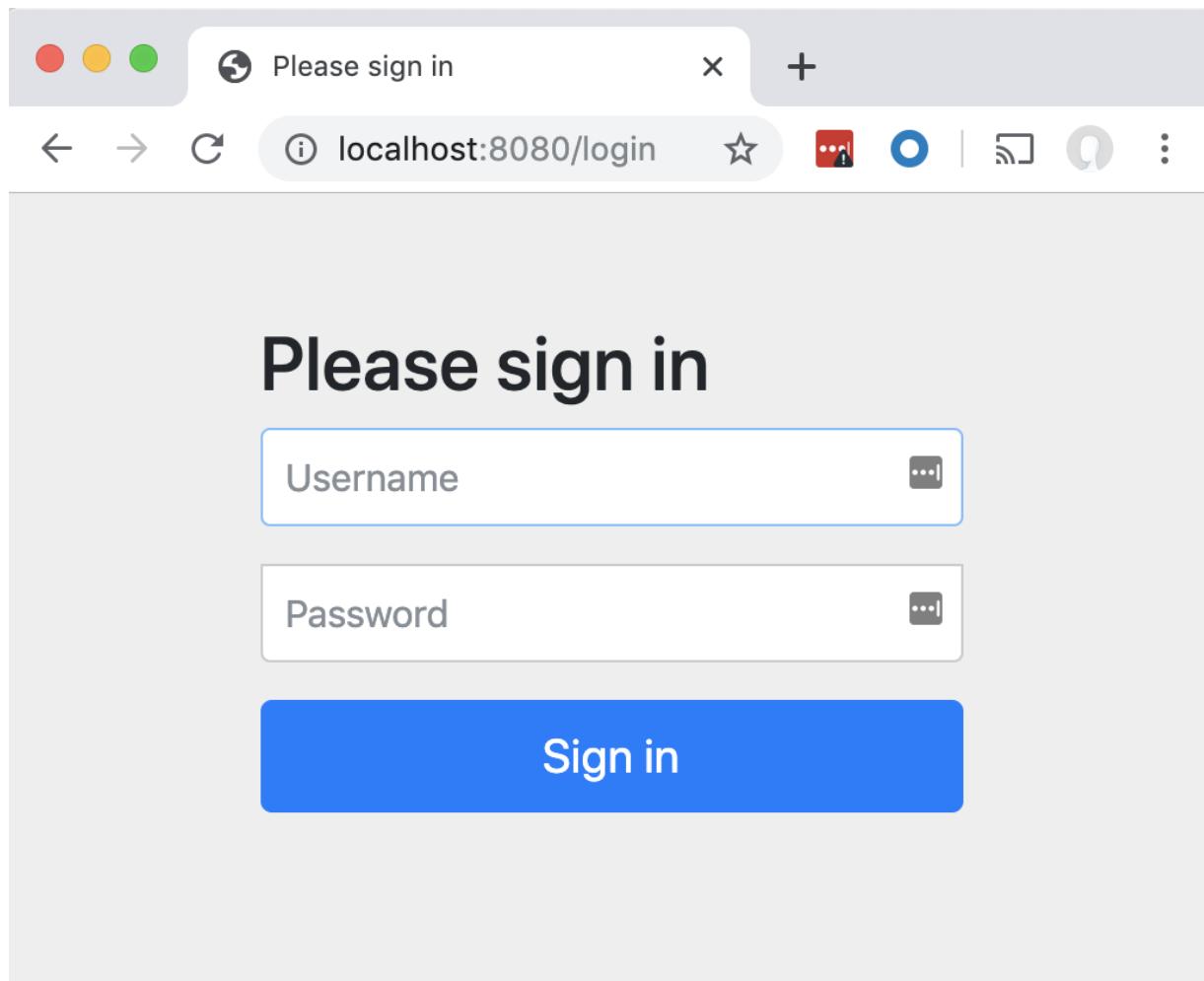
```
...
```

```
2021-02-23 21:31:10.162 INFO 82765 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s):
8080 (http) with context path ''
```

Hmm. A new line never before seen is printed: Using generated security password.

What happens if you visit <http://localhost:8080>?

Spring Boot application with Spring Security on the classpath



- The URL in the browser's address bar is now <http://localhost:8080/login>.
- The page is asking for a username and a password.

Spring Boot will automatically configure Spring Security to lock down your site! (And to avoid a scandalous security scenario where you push your toy app into production, the password isn't "password." Instead, it's randomly generated upon startup and printed out in the console.)

What, exactly, has been activated?

Spring Security offers an extensive, multi-layered approach to security:

- A series of servlet filters are created and registered in the right order.
- Various directives are added to your web pages:
  - Prevent the "wrong" stuff from getting into the browser's caches.
  - Guard against threat vectors like [clickjacking](#), [session fixation](#), [XSS projections](#), and more.
  - Add the right security headers to your responses.
  - Activate [CSRF](#) protections.

Each one of these attack vectors would warrant its own technical article. Suffice it to say, Spring Security is up-to-date on security threats and goes to great strides to make it simpler for you to protect your users.

With Spring Boot, you can secure your application quickly.

While convenient for demos and pitching-to-the-CTO, simply adding Spring Security to your project and pushing things out to production is NOT the way to go. Instead, you need to craft a policy for your application. And that's what the rest of this chapter will dive into.

# Getting real

Demos are nice. As stated in the last section, they are a great way to quickly pitch something to your team leader.

But when it's time to get real, you need something a little more scalable. There are other ways to register all the users that need access to your system. But odds are pretty good that your Security Ops team will want to manage that repository with separate tools.

So what now?

Easy...write a small amount of glue code to connect to the same repository.



You can see all the code in this chapter variant [here](#).

Imagine that the Security team's user management tool is also based on a relational database. It could be any other data store. But the fundamental concept of storing a collection of users along with their passwords and roles is the same.

While Spring Security has multiple ways to hard code a collection of these users, it's simply easier to connect to a live data store. It's not hard to do, and there won't be any growing

pains when you transition from concept to concrete implementation.

To start things off, you need to define your User type. Since you've been using H2 throughout this book, just continue doing so.

### Create a User type

```
@Entity
public class User {

    private @Id @GeneratedValue Integer id; ①
    private String name;
    private String password;
    private @ElementCollection(fetch = FetchType.EAGER) List<String>
    roles;

    protected User() {} ②

    public User(Integer id, String name, String password, List<String>
    roles) { ③
        this.id = id;
        this.name = name;
        this.password = password;
        this.roles = roles;
    }

    public User(String name, String password, List<String> roles) { ④
        this.name = name;
        this.password = password;
        this.roles = roles;
    }

    // just put the rest (getters/setters/etc) here
}
```

- 
- ① Flag the key field using JPA's `@Id` annotation. Define its values using JPA's `@GeneratedValue` annotation.
  - ② A no-argument constructor is critical to support JPA. It has been made protected so that application code doesn't accidentally use it.
  - ③ An all-argument constructor is useful for certain test scenarios.
  - ④ A constructor for all the data fields (but leaving out the `id` field) is handy if you plan to load up several users and only have the data. This lets Spring Data figure out the "next" key value while storing the data.

To access this User data, you need to create a corresponding Spring Data repository definition.

#### Creating a matching repository definition

```
public interface UserRepository extends CrudRepository<User, Integer>
{
    Optional<User> findByName(String name);
}
```

This repository has a single custom finder, `findByName()`. This is critical, as you'll soon see. Spring Security needs to be able to look up a single user based on `username`.



You're free to create any other user fields you need, like email, mailing address, etc. This can all be part of a user's profile. But that is outside the scope of security services. And...you may wish to disentangle such things from security operations anyway. But that's a deeper discussion.

To use this newly minted repository, you need to configure Spring Security. And for that, you need to create a `SecurityConfig` class.

### Configuring Spring Security

```
@Configuration  
public class SecurityConfig {  
  
    ...  
  
}
```

Inside this class, you need to register a bean that uses your custom `UserRepository` to look up `User` objects and transforms them into Spring Security `User` objects.

For Spring Security to pick up and use your custom bean in a servlet-based application, it must implement the `UserDetailsService` interface.

### Custom `UserDetailsService`

```
@Bean
public UserDetailsService userDetailsService(UserRepository
repository) { ①
    return username -> repository.findByName(username) ②
        .map(user -> User.withDefaultPasswordEncoder() ③
            .username(user.getName()) //
            .password(user.getPassword()) //
            .authorities(user.getRoles().toArray(new String[0])) //
            .build())
        .orElseThrow(() -> new UsernameNotFoundException("Could not
find " + username)); ④
}
```

- ① Inject the UserRepository.
- ② UserDetailsService only has one method: `loadUserByUsername(username)` → `UserDetails`. You can define it using a Java 8 lambda. The first step is to use the repository's `findByName` finder method.
- ③ With your custom `Optional<User>` object in hand, you must now transform it, i.e. `map` it to a Spring Security `UserDetails` object. Spring Security's `User` type has a helper method with a nice fluent API that lets you specify the password encoder along with `username`, `password`, and `authorities`. The whole thing is rendered into a `UserDetails` object using the fluent API's `build()` method.
- ④ In the event you get back an `Optional.EMPTY` from the database, throw a `UserNotFoundException`.

Ta dah! You're done. This little bit of glue you wrote will connect your custom JPA repository to Spring Security, allowing user details to be checked as needed.

In case you haven't noticed, you have a custom User type for interacting with H2, and Spring Security has a User type. You could expend a bunch of effort to rename your custom user type to avoid this collision, but the times where you'll be using both types within the same class are few and far between.

The last step in getting things off the ground is to pre-load some test users. To do that, add one more bean to `SecurityConfig`.

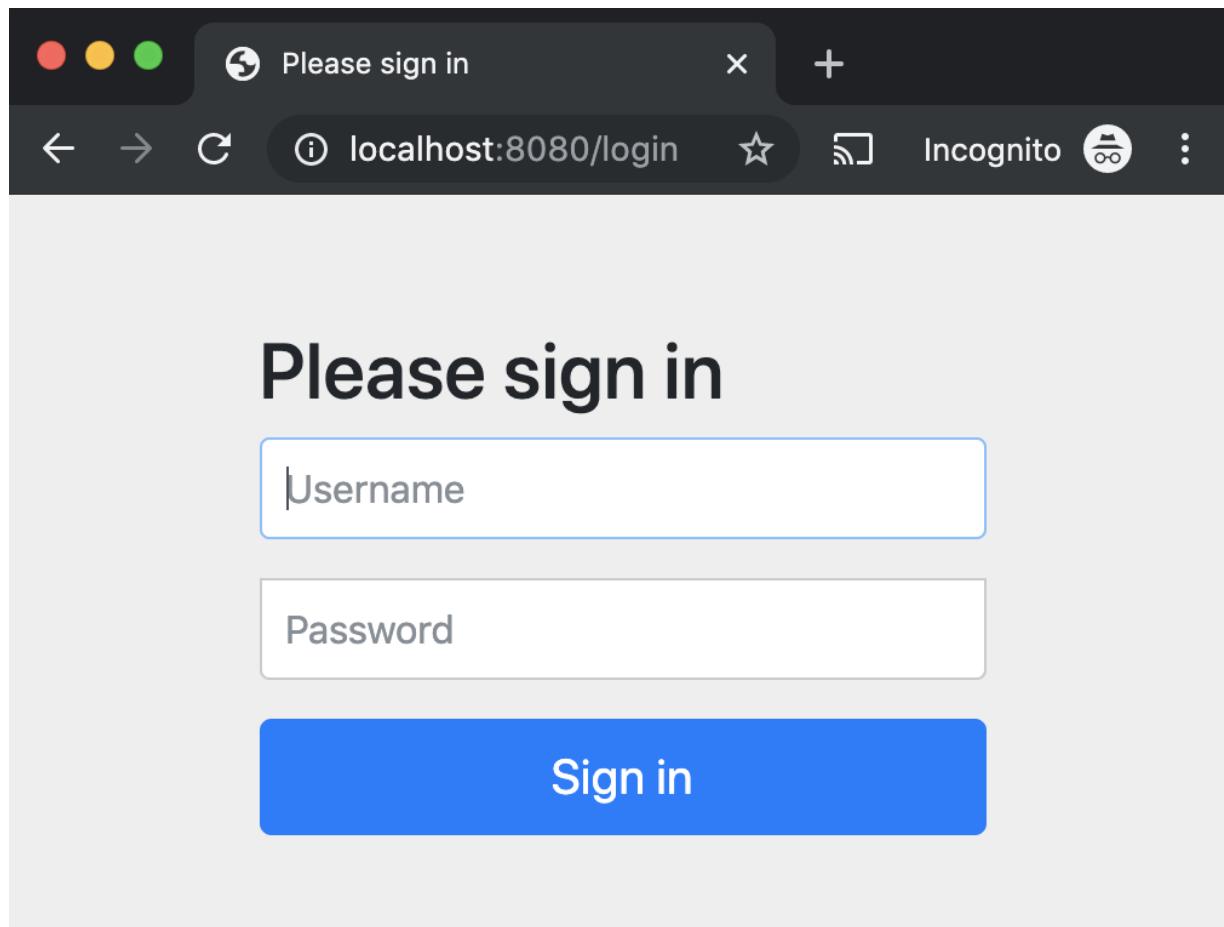
### Adding test users

```
@Bean
CommandLineRunner userLoader(UserRepository repository) {
    return args -> {
        repository.save(new
com.greglturnquist.hackingspringboot.classic.User( //
    "greg", "password", Arrays.asList("ROLE_USER")));
    };
}
```

If you visit Spring Security's reference docs, you'll find other APIs that make it easy to load test users. Frankly, I find pre-loading H2 like this just as easy. Load them into H2 (or whatever data store you're using) and access the data directly from a repository. But the choice is yours.

Fire up the application, and visit the home page (<http://localhost:8080>). You'll immediately get bounced to a login page, as shown below.

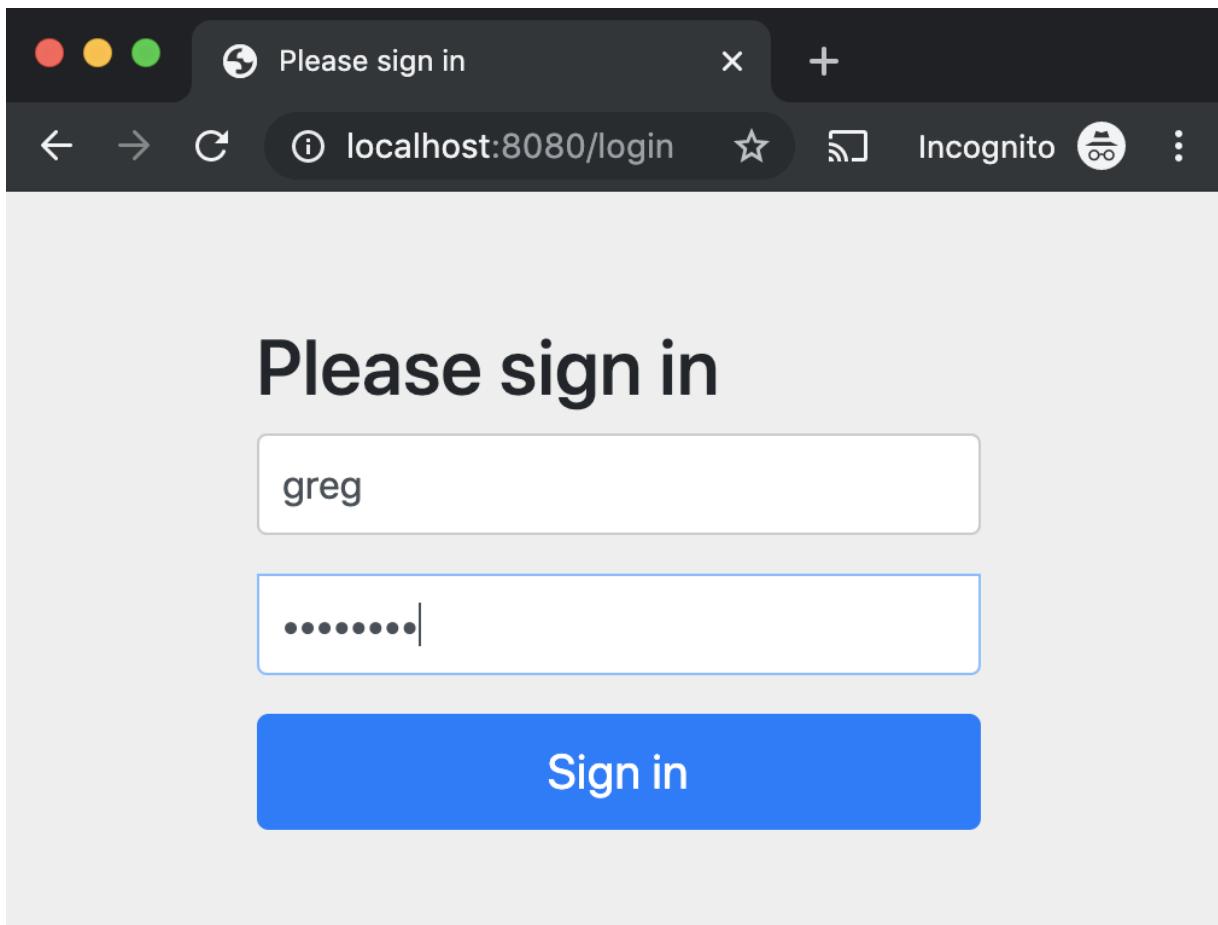
Spring Security's FORM login screen



Don't panic!

You already pre-loaded credentials using the UserRepository inside the SecurityConfig class. Just login as "greg" (or whatever account you have setup in your own version.)

Entering credentials



Click on "Sign in", and it should redirect you back to the home page.

So what, exactly, has been switched on? It's important to understand that Spring Boot is deciding whether or not to apply the `@EnableWebSecurity` annotation provided by Spring Security. And that's about it.

What Spring Security does, by default, when that annotation is applied includes:

- Enable HTTP BASIC, meaning you can use tools like cURL by providing a username/password pair.

- Enable HTTP FORM, meaning that if you visit the site from a browser, you will be served a login page (instead of the browser's default login popup).
- All resources are granted access if the user is authenticated. This means that no further permissions are required to access anything.

Is this good enough?

Well...not really. Granting each authenticated user access to everything is not recommended. You should limit access to what people immediately need. This includes the pages they are permitted to see. But a more subtle thing to include is altering the links users see so they aren't even offered a path to visit those pages.

And we'll explore how to configure this further down in this chapter.

Spring Boot strives to make educated guesses about what you're doing and wire beans accordingly. Revving up an instance of embedded Apache Tomcat and some view resolvers is easy when Spring MVC is on the classpath. However, given the dozens of ways people configure security from app to app, it's almost impossible for Spring Boot to guess what's afoot just because Spring Security is on the classpath.

Back in the 1.x days, Spring Boot attempted to make guesses and offer options. And it was hard to figure out what Boot was doing vs. what Spring Security was doing. Hence, in 2.0+, Spring Boot adopted the position of simply turning on Spring Security, but if it spotted a `springSecurityFilterChain` bean (created when you use `@EnableWebSecurity` directly), then it would back off and let you take over.

In this scenario, configuring a custom `UserDetailsService` still lets the defaults kick in. Spring Security accepts several beans through good old-fashioned dependency injection. But as soon as you start setting the policy (which you'll do in the next section), then Spring Boot will let you take the driver's seat.

## Taking the driver's seat

Spring Security is a curious beast. People need a lot of flexibility when setting up their apps. Some want to lock down sections. Others want to control every page. The login experience varies from site to site.

But certain things have to be done in a certain order or it's not effective. Or flat out falls apart.

And there are few projects that are a better testimony to dependency injection. Spring Security has many injection points including the ability to insert custom filters.

Spring Security strives to register all the critical filters needed to properly secure your application *and* make it possible to register your own filters. It's possible to swap out any of those key filters with a custom one, but that requires deliberate action. Because that's way off the "customize my app" common path, we won't explore that.

But tailoring your security configuration? That's exactly what you'll dig into.



The next section's code can be found [here](#).

The following code shows alterations to apply to `SecurityConfig` in defining a custom policy.

### Writing a custom policy

```
static final String USER = "USER";
static final String INVENTORY = "INVENTORY";

@Override
protected void configure(HttpSecurity http) throws Exception { ①
```

```

http //
    .authorizeRequests() //
    .mvcMatchers(HttpMethod.POST, "/").hasRole(INVENTORY) ②
    .mvcMatchers(HttpMethod.DELETE, "/**").hasRole(INVENTORY) //
    .anyRequest().authenticated() ③
    .and() //
    .httpBasic() ④
    .and() //
    .formLogin() ⑤
    .and() //
    .csrf().disable();
}

```

- ① Create a custom policy by overriding Spring Security's `WebSecurityConfigurerAdapter` and using it to adjust the `HttpSecurity` bean.
- ② Define all your authorization rules, such as HTTP verb/URL patterns and the roles required to attain access. In this case, POST / and DELETE /\*\* require `ROLE_INVENTORY`.
- ③ Any pattern that doesn't match is caught here and simply requires the user be authenticated.
- ④ Enable access via **HTTP BASIC**.
- ⑤ Enable access via **HTTP FORM**.



If you enable **HTTP BASIC**, be sure the connection is secured by something like SSL. Username plus the password are joined by a : and then base64-encoded, which is very easy to reverse. Unless the pipe is secured, it's easy to snoop over the network.

You can expand your set of users (also defined in `SecurityConfig`) to include one with this new `ROLE_INVENTORY`.

## Creating test users with different roles

```
static String role(String auth) {
    return "ROLE_" + auth;
}

@Bean
CommandLineRunner userLoader(UserRepository repository) {
    return args -> {
        repository.save(new
com.greglturnquist.hackingspringboot.classic.User( //
    "greg", "password", Arrays.asList(role(USER))));

        repository.save(new
com.greglturnquist.hackingspringboot.classic.User( //
    "manager", "password", Arrays.asList(role(USER),
role(INVENTORY))));
    };
}
```

Don't forget that USER and INVENTORY were defined in the previous fragment where myCustomSecurityPolicy() is found.



**Roles vs. Authorities** - Spring Security supports the concept of verifying someone has the correct authorization to carry out a certain function. Its simplest, most widely-used implementation is verifying the user's list of **roles** includes a particular value. For example, you can specify that a certain URL requires ROLE\_ADMIN. This entire string is an **authority**.

However, the prefix ROLE\_ has become such a common paradigm, that there are several APIs in Spring Security that simply check if the user has the **role**. While ROLE\_ADMIN would be an **authority**, ADMIN would be the **role** of that authority.

When you start creating rules like this, the first thing you should do...is to test them. And I don't mean spin up the app and do a couple cURL calls. Or log into the site a few times. Those are ancient testing tactics from twenty years ago.

Instead, you should automate testing with and without various authorities.

Write a test case that attempts to add inventory items *without* the property role.

Adding inventory without proper authorization

```
@Test  
@WithMockUser(username = "alice", roles = { "SOME_OTHER_ROLE" }) ①  
void addingInventoryWithoutProperRoleFails() {  
    this.webTestClient.post().uri("/") ②  
        .exchange() ③
```

```
        .expectStatus().isForbidden(); ④  
    }
```

- ① Using Spring Security Test's `@WithMockUser`, have user `alice` with `ROLE_SOME_OTHER_ROLE` carry out this test case.
- ② Using the `WebTestClient`, attempt to POST to the root URI `(/)`.
- ③ Complete the server exchange.
- ④ Verify that the HTTP status is a `403 Forbidden` code.

A `403 Forbidden` result show that while the user was *authenticated*, they weren't *authorized* to perform the web call.

Having verified that users with the wrong role won't be granted access, now verify that users with the right role can carry out this operation.

### Adding inventory with proper authorization

```
@Test  
@WithMockUser(username = "bob", roles = { "INVENTORY" }) ①  
void addingInventoryWithProperRoleSucceeds() throws  
InterruptedException {  
    this.webTestClient //  
        .post().uri("/") //  
        .contentType(MediaType.APPLICATION_JSON) ②  
        .bodyValue("{" + ③  
            "\"name\": \"iPhone 11\", " + //  
            "\"description\": \"upgrade\", " + //  
            "\"price\": 999.99" + //  
            "}") //  
        .exchange() //  
        .expectStatus().isFound(); ④  
  
    assertThat(this.repository.findByName("iPhone  
11")).hasValueSatisfying(item -> { ⑤
```

```
        assertThat(item.getDescription()).isEqualTo("upgrade"); ⑥
        assertThat(item.getPrice()).isEqualTo(999.99);
    });
}
```

- ① This time, user bob has authority ROLE\_INVENTORY.
- ② Send a Content-Type: application/json request header to signal a JSON document is being sent for data.
- ③ Transmit the new piece of inventory using JSON.
- ④ After doing the exchange with the server, verify a 200 Ok response header is sent back.
- ⑤ Use the injected ItemRepository to query H2 after the operation has completed for the new Item.
- ⑥ Verify the other fields of the newly created Item. (No need to verify name since you already searched by that!)

Simply put, you are submitting an HTTP **POST** with a bit of JSON, and then verifying the results by peeking in the database.



For every security rule you write, it makes sense to write at least *two* test cases—one for the success path and one for a failing path. So far, you've covered verifying the **POST** operation. It's left as an exercise for you to verify the **DELETE** operation.

## Tapping into user context

An important aspect of bringing in security management is the fact that you have access to the current user's details.

Throughout this book, presuming you've followed the examples, the name of the cart has simply been My Cart.

With access to user login details, you can suddenly support a different cart for every user. The code below shows how to do just that.

Making the shopping cart user-specific

```
@GetMapping  
String home(Authentication auth, Model model) { ①  
    model.addAttribute("items", //  
        this.inventoryService.getInventory());  
    model.addAttribute("cart", //  
        this.inventoryService.getCart(cartName(auth)) //  
            .orElseGet(() -> new Cart(cartName(auth)))); ②  
    model.addAttribute("auth", auth); ③  
  
    return "home";  
}
```

- ① Add Authentication as another parameter to the `home()` method so that Spring Security will extract it from the **Servlet context**.
- ② Invoke `cartName()` (shown further down) to translate it into a cart's id.
- ③ Supply the Authentication object to the template to contextualize the web page.

Before diving into adjustments on the web page, it's important to flesh out exactly what this `cartName` function does. It's not complex.

Creating a cart name based on user details

```
private static String cartName(Authentication auth) {  
    return auth.getName() + "'s Cart";  
}
```

By moving this logic into a simple, static function, you've centralized the source of truth for a given user's cart.



I'm a big fan of minimizing the exposure of methods and variables. This means declaring things private. At least until it comes time to share. The function above that turns an Authentication into a cart name is private because no one else needs access to this. But if you take the code, do some tweaking, add more services, and need to widen access, by all means do it!

There is also incredible benefit to adding the Authentication object to the template's model. It makes it possible to show users contextual information.

Add this HTML fragment to the `home.html` template's `<body>` tag, right at the top.

Sprinkling in some user context on the web page

```
<table>  
    <tr>  
        <td>Name:</td>  
        <td th:text="${auth.name}"></td>  
    </tr>  
    <tr>  
        <td>Authorities:</td>
```

```
<td th:text="${auth.authorities}"></td>
</tr>
</table>
<form action="/logout" method="post">
    <input type="submit" value="Logout">
</form>
<hr/>
```

This fragment renders a `<table>` that shows the user's name as well as a list of authorities. Not recommended for production, but it's quite handy to ensure you have the needed data in the web layer.

There is also a **Logout** button, making it easy to test various scenarios.

Finally, a nice horizontal line (`<hr/>`) is drawn. No need for slick CSS.

Restart the application and navigate to <http://localhost:8080>. After logging in, you should see new user context at the top.  
Logging in and seeing user details

# Welcome to Hacking with Spring Boot!

## Inventory Management

<b>Id</b>	<b>Name</b>	<b>Price</b>	
1	Alf alarm clock	19.99	<a href="#">Add to Cart</a>
2	Smurf TV tray	24.99	<a href="#">Add to Cart</a>

## My Cart

<b>Id</b>	<b>Name</b>	<b>Quantity</b>
-----------	-------------	-----------------

The newly-added HTML fragment nicely renders the user, their roles, and the Logout button.

The last bit of code to tweak are the web functions for adding and removing item's from the user's cart. These must also leverage the `cartName()` function.

Last tweaks to controller code

```
@PostMapping("/add/{id}")
String addToCart(Authentication auth, @PathVariable Integer id) {
```

```
        this.inventoryService.addItemToCart(cartName(auth), id);
        return "redirect:/";
    }

    @DeleteMapping("/remove/{id}")
    String removeFromCart(Authentication auth, @PathVariable Integer id)
    {
        this.inventoryService.removeOneFromCart(cartName(auth), id);
        return "redirect:/";
    }
}
```

In both methods, `Authentication` has been added as a method parameter. The `cartName()` function is key in finding a user's cart before making changes.

With this simple change to making a given cart based on user details, you've migrated the entire application from one global shopping cart to a multi-user system, capable of storing literally millions of shopping carts with ease.



Now you have to handle the problem of abandoned shopping carts!

These adjustments set us up for the next section—method-level security.

## Method-level security

Having put in a basic amount of security, it's time to point out some of its issues. HTTP verb/URL rules like `mvcMatchers(POST, "/").hasRole(...)` grants you fine-grained control, but it has its limitations:

- Changes in the controller class could require changes in your security policy.
- As you add more controllers and thus more lines to your `HttpSecurity` bean, this could get unwieldy.
- What about chunks of code that need role-based security, but aren't directly linked to web endpoints?

That's where method-level security steps in.

By applying a Spring Security annotation *directly on the method*, you can secure things directly where your business logic is located. Avoid having to manage security rules for the URLs of a dozen controllers and instead, keep the domain of security right alongside the logic of business.

Now to poke around with method-level security, you need something a little more substantive than a couple of web methods. So for this section, you'll add a REST API using Spring HATEOAS (see *Chapter 6, Building APIs with Spring Boot*).



This section's method-based code can be found [here](#).

You must opt into method-level security. It is NOT activated by default. You only need apply the following annotation (ideally to your security configuration class).

### Enabling method-level security

```
@Configuration  
@EnableGlobalMethodSecurity(prePostEnabled = true)  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
    ...  
}
```

This variant of `SecurityConfig` shows `@EnableGlobalMethodSecurity` being applied with the `prePostEnabled` optional enabled. For servlet-based apps, it's important to use this version (not the reactive one!) And Spring Security has supported multiple variants of annotations. The "pre-post" version is the most sophisticated and will be the one used.

The first step in moving to a method-based approach is to remove the `mvcMatchers()` clauses used earlier in this chapter.

### Simplified security policy

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http //  
        .authorizeRequests(registry -> registry //  
            .anyRequest().authenticated() //  
            .httpBasic() //  
            .and() //  
            .formLogin() //
```

```
        .and() //
        .csrf().disable();
}
```

This is close to the default security policy. The only difference is disabling CSRF.

Having removed authorization rules, it's time to switch your attention toward writing an `ApiItemController`. This code was first crafted in *Chapter 6, Building APIs with Spring Boot*, where you constructed hypermedia-based APIs.

Instead of repeating all that, just focus on writing a method that can save a new `Item` object—but only if the user is authorized! Adding a new item...if authorized

```
@PreAuthorize("hasRole('" + SecurityConfig.INVENTORY + "')") ①
@PostMapping("/api/items/add") ②
ResponseEntity<?> addNewItem(@RequestBody Item item, Authentication
auth) { ③
    Item savedItem = this.repository.save(item);

    EntityModel<Item> newModel = findOne(savedItem.getId(), auth);

    return ResponseEntity.created(newModel //
        .getRequiredLink(IanaLinkRelations.SELF) //
        .toUri()).build();
}
```

① `@PreAuthorize` is Spring Security's key annotation for securing things at the method level. The Spring Security SpEL expression asserts whether or not the invoking user has authority `ROLE_INVENTORY` *before* invoking the

method. (By the way, INVENTORY is just a constant shown earlier in this chapter with the value of "INVENTORY".)

- ② `@PostMapping("/api/items/add")` signals this method to respond to HTTP POST calls to `/api/items/add`.
- ③ This method also asks for a copy of the invoking user's `Authentication` object. This can be done if your method, for whatever reason, needs to see the current user's security context. (HINT: You'll leverage it a little further down in this section!)



Earlier I mentioned the value of declaring things `private`. Another option is to consider using Java's default visibility rules. Remember creating `static final String USER = "USER"` inside `SecurityConfig`? The lack of `public` makes `USER` a **package private** field and thus only visible to the classes in the same package. When you're in the early days of application development, it's perfect. If your app grows into something bigger, the amount of coupling from external components is vastly reduced. And the parts you do expose becomes a deliberate choice.

The delete operation needs to be updated as well.

Deleting an existing item with method-level security

```
@PreAuthorize("hasRole('" + SecurityConfig.INVENTORY + "')")
@DeleteMapping("/api/items/delete/{id}")
ResponseEntity<?> deleteItem(@PathVariable Integer id) {
    this.repository.deleteById(id);
    return ResponseEntity.noContent().build();
}
```

- This method has the same `@PreAuthorize` annotation, indicating it also requires the invoking user to hold `ROLE_INVENTORY`.
- The `@DeleteMapping` annotation will connect the method to HTTP DELETE calls.
- It also has a templated URI responding to `/api/items/delete/{id}`.
- The rest of the code simply carries out the H2 delete operation, and then returns an **HTTP 204 No Content** response, signaling success.



Shouldn't these methods just re-use the same URIs like `/api/items` and `/api/items/{id}` for POST and DELETE? Yes, it's a common convention to have the same endpoint that fetches data also respond to other HTTP verbs. However, in a HAL document, those methods would result in the same set of links getting rendered. Since HAL doesn't show HTTP verbs, it doesn't provide the best demo. So I shifted these operations to slightly different URIs.

`@PreAuthorize` is Spring Security's heavy hitter when it comes to method-level security. You can feed it rather complex expressions even utilizing arguments in the method.

It's also possible to make this decision *after* the method has been called using `@PostAuthorize`. This is an option if the return

value carries some key contextual ingredient you need to make the decision. Simply refer to the return value using `returnObject` in your Spring Security SpEL expression. Just don't forget, making a database call and then voiding the whole thing isn't free.

If you are returning a collection of results, you can also filter the results using `@PostFilter`. This gives you the ability to filter out data results the current user isn't authorized to see. While handy, it can be taxing to fetch a lot of data that ultimately gets filtered out. That's why Spring Data also supports the ability to embed `Authentication` objects into queries to filter directly inside the database.



Spring Security/Spring Data integration is *only* supported when using the `@Query` annotation.

Now don't forget! As you write these security policies, it's good to write some test cases to verify things are working properly. Here's a test case where a mock user attempts to add a new `Item` object *without* proper authorization.

#### Testing unauthorized users

```
@Test  
 @WithMockUser(username = "alice", roles = { "SOME_OTHER_ROLE" }) ①  
 void addingInventoryWithoutProperRoleFails() {  
     this.webTestClient //
```

```

    .post().uri("/api/items/add") ②
    .contentType(MediaType.APPLICATION_JSON) //
    .bodyValue("{" + //
        "\"name\": \"iPhone X\", " + //
        "\"description\": \"upgrade\", " + //
        "\"price\": 999.99" + //
        "}") //
    .exchange() //
    .expectStatus().isForbidden(); ③
}

```

- ① Using Spring Security's `@WithMockUser` annotation, you can arm this mock user with a role known to NOT work.
- ② Issue a **POST** to the `/api/items/add` URI with all the proper inputs.
- ③ Verify the resulting status is an HTTP **403 Forbidden**. This truly signals that you had valid **authentication** and were let in. But you didn't possess the right authorization to carry out the operation. And with no `mvcMatchers()` rules in sight, you have verified proper migration to method-level security.

You have successfully verified the "bad" code path, where a user lacks the proper role. Now it's time to write a test case for the "good" path.

### Testing properly authorized users

```

@Test
@WithMockUser(username = "bob", roles = { "INVENTORY" }) ①
void addingInventoryWithProperRoleSucceeds() {
    this.webTestClient //
        .post().uri("/api/items/add") ②
        .contentType(MediaType.APPLICATION_JSON) //
        .bodyValue("{" + //
            "\"name\": \"iPhone X\", " + //
            "\"description\": \"upgrade\", " + //

```

```
    "\"price\": 999.99" + //  
    "}") //  
.exchange() //  
.expectStatus().isCreated(); ③  
  
    assertThat(this.repository.findByName("iPhone  
X")).hasValueSatisfying(item -> { ④  
        assertThat(item.getDescription()).isEqualTo("upgrade");  
        assertThat(item.getPrice()).isEqualTo(999.99);  
    });  
}
```

- ① Set up mock user "bob" with the "INVENTORY" role.
- ② Post the same request as before.
- ③ Verify the exchange yields an **HTTP 201 Created** status (success!)
- ④ Verify the new item was properly stored in the database.

These test cases make it a snap to verify your security profile is configured properly. And by coding at least one test case for each "good" code path and for each "bad" code path, you can generate high confidence that the policy is working as designed.

Will this generate a lot of work if you alter security policies? Yes. But that's good. If security policies change, it can impact a lot of things, and you need to be alerted to that.



You have tested that adding a new Item to inventory is properly secured. It's left as an exercise to verify the **DELETE** operation also works as expected. Don't forget—at least two test cases!

Rule #1 in security land—don’t allow people to execute operations for which they lack proper authority. You’ve just done that. Only users with ROLE\_INVENTORY will be allowed to alter the system’s inventory.

But it’s just as important to pay heed to Rule #2 in security land —don’t show a user anything that will cause them to run into Rule #1. From a hypermedia perspective, don’t include links they can’t navigate.

To exercise this, let’s examine that `findOne` operation meant to show a hypermedia record and see if you can conditionalize some of its links.

#### Conditional links based on authorization

```
private static final SimpleGrantedAuthority ROLE_INVENTORY = //  
    new SimpleGrantedAuthority("ROLE_" + SecurityConfig.INVENTORY);  
  
 @GetMapping("/api/items/{id}")  
 EntityModel<Item> findOne(@PathVariable Integer id, Authentication  
 auth) {  
     ApiItemController controller = methodOn(ApiItemController.class);  
  
     Link selfLink = linkTo(controller.findOne(id, auth)).withSelfRel();  
  
     Link aggregateLink = linkTo(controller.findAll(auth)) //  
         .withRel(IanaLinkRelations.ITEM);  
  
     Links allLinks; ①  
  
     if (auth.getAuthorities().contains(ROLE_INVENTORY)) { ②  
         Link deleteLink =  
             linkTo(controller.deleteItem(id)).withRel("delete");
```

```

    allLinks = Links.of(selfLink, aggregateLink, deleteLink);
} else { ③
    allLinks = Links.of(selfLink, aggregateLink);
}

return this.repository.findById(id) ④
    .map(item -> EntityModel.of(item, allLinks)) //
    .orElseThrow(() -> new IllegalStateException("Couldn't find
item " + id));
}

```

This method contains quite a bit. If you glossed over *Chapter 6, Building APIs with Spring Boot*, you may want to revisit it. For now, peek at the security-specific parts:

- ① You are trying to assemble a Spring HATEOAS link container, `Links`.
- ② Check that the user possesses `ROLE_INVENTORY`. If so, include the **DELETE** link along with the **self** and aggregate root links.
- ③ If the user does NOT possess `ROLE_INVENTORY` role, *only* include the **self** and aggregate root links.
- ④ Now fetch the desired `Item` object from the data store, and transform that into a Spring HATEOAS `EntityModel` container, with all the `Links` provided.

Now that little method packs a punch! In fact, if you didn't comprehend everything, I suggest you go back and re-read it. Of course this isn't the only way to use `Authentication` objects to customize output. What scenarios can you think of that would need to provide an either-or situation?

Remember the thing about writing test cases? With methods that produce HAL hypermedia documents, it's not hard to verify

they contain what's expected. Before you can write a test case, you first need to configure WebTestClient to consume these hypermedia documents.

## Configuring WebTestClient with hypermedia navigation

```
@SpringBootTest()
@EnableHypermediaSupport(type = HAL) ①
@AutoConfigureMockMvc
public class ApiItemControllerTest {

    WebTestClient webTestClient;

    @Autowired ItemRepository repository;

    @Autowired HypermediaWebTestClientConfigurer webClientConfigurer;
    ②

    @BeforeEach
    void setUp(@Autowired MockMvc mockMvc) { ③
        this.webTestClient = MockMvcWebTestClient //
            .bindTo(mockMvc) //
            .apply(webClientConfigurer) ④
            .build();
    }

}
```

- ① Somewhere, you need to `@EnableHypermediaSupport` for HAL. The application itself will apply that annotation when you put Spring HATEOAS on the classpath. For this test case, apply the annotation.
- ② Inject Spring HATEOAS's `HypermediaWebTestClientConfigurer`.
- ③ Inject the autowired `MockMvc` and use it to create a `WebTestClient`.
- ④ Apply `webClientConfigurer` to the `webTestClient`, and it will register support for any activated hypermedia types.

Well, with things all ready to go, proceed to write your first test case that consumes hypermedia.

## Testing hypermedia

```
@Test
@WithMockUser(username = "alice", roles = { "INVENTORY" })
void navigateToItemWithInventoryAuthority() {

    // Navigate to the root URI of the API.
    RepresentationModel<?> root = this.webTestClient.get().uri("/api")
    //
    .exchange() //
    .expectBody(RepresentationModel.class) //
    .returnResult().getResponseBody();

    // Drill down to the Item aggregate root.
    CollectionModel<EntityModel<Item>> items = this.webTestClient.get()
    //
    .uri(root.getRequiredLink(IanaLinkRelations.ITEM).toUri()) //
    .exchange() //
    .expectBody(new CollectionModelType<EntityModel<Item>>() {}) //
    .returnResult().getResponseBody();

    assertThat(items.getLinks()).hasSize(2);
    assertThat(items.hasLink(IanaLinkRelations.SELF)).isTrue();
    assertThat(items.hasLink("add")).isTrue();

    // Find the first Item...
    EntityModel<Item> first = items.getContent().iterator().next();

    // ...and extract it's single-item entry.
    EntityModel<Item> item = this.webTestClient.get() //
        .uri(first.getRequiredLink(IanaLinkRelations.SELF).toUri()) //
        .exchange() //
        .expectBody(new EntityModelType<Item>() {}) //
        .returnResult().getResponseBody();
```

```
        assertThat(item.getLinks()).hasSize(3);
        assertThat(item.hasLink(IanaLinkRelations.SELF)).isTrue();
        assertThat(item.hasLink(IanaLinkRelations.ITEM)).isTrue();
        assertThat(item.hasLink("delete")).isTrue();
    }
```

The first `WebTestClient` does a **GET** call to access the root API's collection of links. Since this is nothing but links, you can extract it into a `RepresentationModel<?>` object.

From here, ask for the **item** link and extract its URI. Convert the aggregate root's response into a `CollectionModel<EntityModel<Item>>`. In case you missed it, this is a collection of `Item` entities. The collection has links as does each entity.

Since this test case involves a user with proper authorization, verify it has both a **self** link as well as an **add** link.

Grab the first entry in the collection and navigate to it with one more web call.

Verify it contains all expected links including the **delete** link.

Whew! That's a lot.

If you wish to write the test case verifying how the hypermedia looks when the user does NOT have `ROLE_INVENTORY`, have at it.



It's left as an exercise to conditionally render an **add** link in the `findAll()` web method.

Things are looking pretty good. You now have the means to apply fine-grained security controls. You can even alter the appearance of outputs based on the user's security profile!

What other ways can you think of that use custom user context to alter APIs and HTML?

## OAuth security

With the rise of social media networks, a new security issue was discovered.

Companies built widely-used websites—but also APIs to build your own apps. And waves of 3rd-party applications emerged. The social media networks were excited. More apps—more eyeballs on their site.

But people were having to enter their credentials into the 3rd-party applications to access the social media networks.

And that started to annoy users. If they altered their credentials on the site itself, they had to update their apps as well. And that wasn't the end of it. If *any* of the 3rd-party systems got hacked OR the main site itself, everyone was compromised. Shared

credentials has always been bad. Building an enterprise on top of direct usage was doomed.

Thus arose the concept of **OAuth**.

OAuth is an open protocol that provides "secure delegated access." In practical terms, for your 3rd-party app to access the social media network, it doesn't store credentials. Instead, it launches a login page on the social media site itself. The user enters their credentials directly, and a secure token is handed back to the application.

The application, instead of managing credentials, only has to hold onto the token. These tokens also come with key features like expiration, refreshing, and other key features already baked in.



The OAuth spec has gone through some revisions. It started simple, and expanded in options. However, that flexibility came at a price. Every provider could do OAuth a little differently. Today, the adopted industry standard is **OpenID Connect 1.0** or OIDC. In essence just about every web site you know of that has OAuth support does it through OIDC.

So the question arises. "How do I configure my Spring application to login using Google or Facebook?"

You can dive in right here and find out!



This section's code can be found [here](#).

You need to add the following dependencies to your project (instead of `spring-boot-starter-security`).

#### OAuth-based dependencies

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

This has three dependencies:

- `spring-security-config` - to use Spring Security configuration annotations and types in your controllers
- `spring-security-oauth-client` - to interact with OAuth providers as a client

- `spring-security-oauth2-jose` - support for **Javascript Object Signing and Encryption** or JOSE

Next, you need to register your new application with Google.

1. Visit

<https://developers.google.com/identity/protocols/OpenIDConnect>.

2. Open the **Credentials page**.

3. If you haven't done so already, create your project's OAuth 2.0 credentials by clicking + **CREATE CREDENTIALS** > **OAuth client ID**, and providing the information needed to create the credentials. (HINT: This is a web application.)

4. Look for the client ID you just declared in the **OAuth 2.0 Client IDs** section and click on it.

After you have gathered your client id and client secret, switch back to your IDE. Throughout this book, you've seen examples of configuring application property settings using `application.properties`. Did you know Spring Boot also supports using a YAML-based version of these files. Yup, you can also have `application.yml` as your means to configure things.

It's really a matter of personal taste. The YAML format requires an IDE or editor that will put in the proper spaces. But it is also handy if you lots of properties at the same level. In particular, it's quite when configuring OAuth settings. So let's close out this

chapter using application.yml instead of application.properties.

Configure your application.yml like this.  
Plugging in OAuth credentials to your application

```
spring:  
  security:  
    oauth2:  
      client:  
        registration:  
          google:  
            client-id: 76769redactedoogleusercontent.com  
            client-secret: ovj6b_va5fEB-EWEcE3FGOK6
```

You should notice two properties configured via YAML that have google buried in the middle. This is an example of an OAuth provider that Spring Security has already configured most of the details. You simply provide the client-id and client-secret.

Of course your id and secret will be different.

Spring Security comes with pre-built support for Google, GitHub, Facebook, and Okta. (Just check out CommonOAuth2Provider to see exactly what providers are covered and their complete settings.)

With this hooked in, you'll need to tweak that code in the HomeController that fetches the user's cart when they visit the

site's home page.

## OAuth support in your controller

```
@GetMapping  
String home( //  
    @RegisteredOAuth2AuthorizedClient OAuth2AuthorizedClient  
    authorizedClient,  
    @AuthenticationPrincipal OAuth2User oauth2User, Model model) { ①  
    model.addAttribute("items", this.inventoryService.getInventory());  
    model.addAttribute("cart",  
        this.inventoryService.getCart(cartName(oauth2User)) ②  
        .orElseGet(() -> new Cart(cartName(oauth2User))));  
  
    // Fetching authentication details is a little more complex  
    model.addAttribute("userName", oauth2User.getName());  
    model.addAttribute("authorities", oauth2User.getAuthorities());  
    model.addAttribute("clientName", //  
        authorizedClient.getClientRegistration().getClientName());  
    model.addAttribute("userAttributes", oauth2User.getAttributes());  
  
    return "home";  
}
```

- ① Instead of asking for a simple `Authentication` object, you can ask for an `OAuth2AuthorizedClient` and/or an `OAuth2User`. This gives you details about the user that logged in as well as the client handling the user. The `@RegisteredOAuth2AuthorizedClient` and `@AuthenticationPrincipal` annotations are needed to help resolve these attributes in a controller method.
- ② If you'll notice, the function to find the cart's name now reads `cartName(oauth2User)`. Switching from `Authentication` to `OAuth2User` will also require another change.

The function you wrote earlier to transform a username into a cart name must be upgraded!

### Basing cart name on OAuth2 user

```
private static String cartName(OAuth2User oAuth2User) {  
    return oAuth2User.getName() + "'s Cart";  
}
```

You also need to update `addToCart` and `removeFromCart` as well to use these annotations instead of `Authentication` objects. It's left as an exercise for the reader to make these adjustments.

To show off what you now have access to, adjust the top of your HTML template with these extra bits (including the `sec` namespace definition).

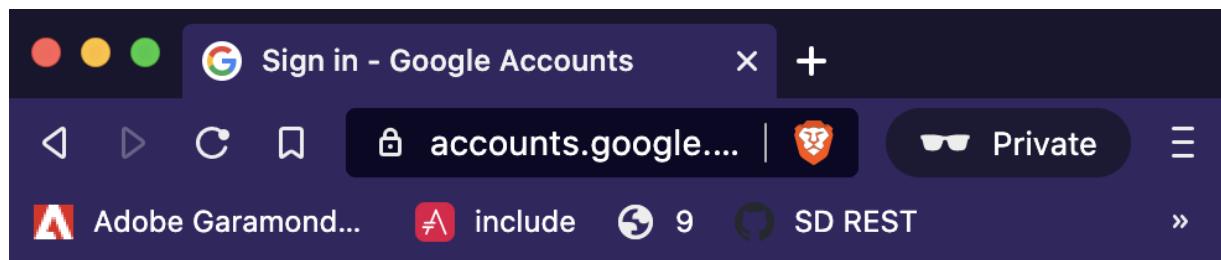
### Putting OAuth details in your web page

```
<html lang="en" xmlns:th="http://www.thymeleaf.org"  
      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-  
      springsecurity5">  
  
<div sec:authorize="isAuthenticated()">  
    <table>  
      <tr>  
        <td>User:</td>  
        <td><span sec:authentication="name"></span></td>  
      </tr>  
      <tr>  
        <td>Authorities:</td>  
        <td th:text="${authorities}"></td>  
      </tr>  
      <tr th:each="userAttribute : ${userAttributes}">  
        <td th:text="${userAttribute.key}"></td>
```

```
        <td th:text="${userAttribute.value}" />
    </tr>
</table>
<form action="#" th:action="@{/logout}" method="post">
    <input type="submit" value="Logout"/>
</form>
</div>
```

If you fire up the application and visit <http://localhost:8080>, things will look *very* different!

Forwarded to Google's login screen



 Sign in with Google

## Sign in

to continue to [Hacking with Spring Boot](#)

Email or phone

[Forgot email?](#)

To continue, Google will share your name, email address, language preference, and profile picture with Hacking with Spring Boot.

[Create account](#)

[Next](#)

Yup. You got redirected to Google's login page.  
After entering credentials, rerouted back to the application

The screenshot shows a browser window with the title "Hacking with Spring Boot - Geti" and the URL "localhost:8080/#". The page displays a user profile with the following details:

User:	104322183681776349740
Authorities:	[ROLE_USER, SCOPE_https://www.googleapis.com/auth/userinfo.email, SCOPE_https://www.googleapis.com/auth/userinfo.profile, SCOPE_openid]
at_hash	JzIz_CWkTBqLyS7VTJ9i_Q
aud	[76769467445-cg244gaa3heg88phvlsfbkehj96opeh.apps.googleusercontent.com]
sub	104322183681776349740
email_verified	true
azp	76769467445-cg244gaa3heg88phvlsfbkehj96opeh.apps.googleusercontent.com
iss	https://accounts.google.com
exp	2021-03-03T22:52:39Z
nonce	B8jAlmDphl2dSEXS6d-Dy_n6yLPO4bHrD-Eqf1akWWM
iat	2021-03-03T21:52:39Z
email	greg.l.turnquist@gmail.com

[Logout](#)

# Welcome to Hacking with Spring Boot!

## Inventory Management

<b>Id</b>	<b>Name</b>	<b>Price</b>	
1	Alf alarm clock	19.99	<a href="#">Add to Cart</a>
2	Smurf TV tray	24.99	<a href="#">Add to Cart</a>

## My Cart

<b>Id</b>	<b>Name</b>	<b>Quantity</b>
-----------	-------------	-----------------

At the top, you'll see similar entries as shown earlier including user and authorities. However, the user value is different. In this case, it's a gigantic number. That's because the application itself is technically the user, not whomever just logged in. And

the authorities have been extended to include what are known in OAuth as "scopes".

A key reason to adopt OAuth 2 is to delegate user management elsewhere. Given the number of security compromises occurring, it's not unwise to let someone else manage credentials. That opens the door to multiple options, be it Google, Facebook, Okta, GitHub, or someone else.

Each one has their tradeoffs.

- Google and Facebook are probably the most universal, meaning that most of your users probably already have an account.
- GitHub is more aimed at the developer community.
- Okta is quite handy if you still need fine-grained user control, but would like to offload some of the complexities and focus on operations.



What if you wanted to login using Apple? Or Twitter? Or any other OAuth Connect ID provider?

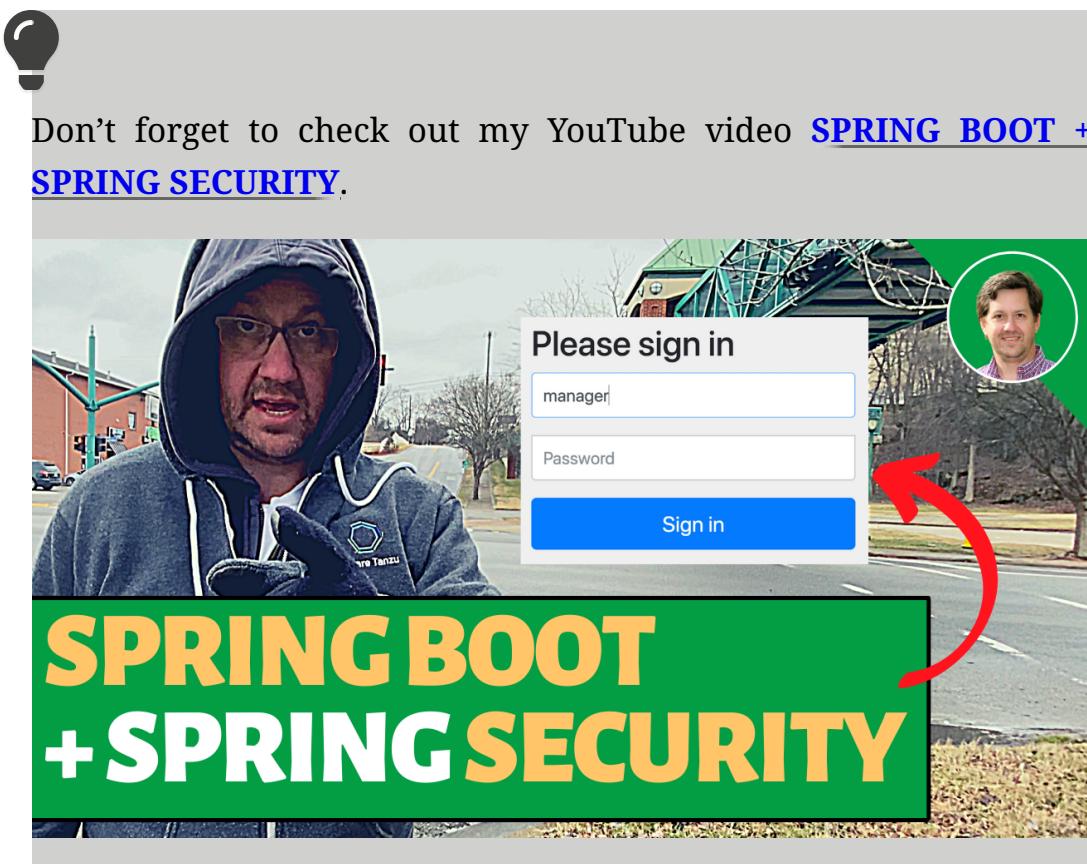
You can do that as well. It just takes a little more effort since they aren't prebaked into Spring Security's OAuth support. If you check the OAuth service you wish to integrate with, as well as the source code for Spring Security's support for Google, it shouldn't be very difficult to align Spring Security's OIDC properties with the values needed to integrate.

Something to realize when you delegate user management elsewhere is that instead of declaring a bunch of roles or authorities, you have to deal with "scopes." You can think of this as simply another type of authority, but prefixed with "SCOPE\_". The biggest tradeoff is asking whether or not you need custom scopes.

If not, then any of these will work. But if you need the ability to assign custom groups or scopes, then you should probably go with something like Okta where you can actually dispense these to various users. The scopes provided by Facebook and Google are aimed at allowing you to interact with their various APIs, meaning there is no room to declare your own.

With all this in hand, you are prepped to go and configure your system to do anything. You can use Google or Facebook if you need universal access for your users with no need for custom

scopes. If your users are mostly developers, you can use GitHub. And if you need easy-to-configure 3rd-party management, but still need extensible role management, then Okta is probably your best bet.



All of these things will make it possible for you to configure your application to provide strong security with your Spring Boot application.

## Summary

In this chapter you:

- Created a demo application by simply adding Spring Boot's security starter.
- Crafted a Spring Data repository to delegate user management to a database.
- Customized your configuration and plugged in URL security rules.
- Leveraged method-level security to apply fine-grained access controls.
- Integrated with Google to shift the need for user management to a 3rd-party.

I hope you have enjoyed getting your feet wet with *Hacking with Spring Boot 2.4: Classic Edition*.

Please [leave an honest review](#)! It's one of the biggest things you can do to help me out.

If you are interested in more, stay tuned.

[Join my email list](#) and you can catch my technical newsletters.

Don't forget to [subscribe to my YouTube channel, Spring Boot Learning](#), the channel where you learn about Spring Boot and have fun doing it.

Thanks,

*Greg L. Turnquist*

# ABOUT THE AUTHOR

---

**Greg L. Turnquist** works on the Spring team as a principal developer at VMware. He is a committer to Spring HATEOAS, Spring Data, Spring Boot, R2DBC, and Spring Session for MongoDB. He founded the Nashville JUG in 2010 and hasn't met a Java app (yet) that he doesn't like.

Catch all of Greg's future works by [signing up to his newsletter](#). Get a FREE technical handout as a bonus!

Be sure to [subscribe to his YouTube channel, Spring Boot Learning](#), the channel where you learn Spring Boot and having fun doing it.

Also enjoy his other published works, technical as well as fiction:

*Technical*

[\*Hacking with Spring Boot 2.3: Reactive Edition\*](#)

[\*Learning Spring Boot 2.0 2nd Edition\*](#)

[\*Learning Spring Boot Video\*](#)

[\*Learning Spring Boot\*](#)

*Python Testing Cookbook*

*Spring Python 1.1*

*Fiction*

*Darklight*

*The Job: A Darklight Chronicle*