Spring Web Essentials

Building modern web services with Spring Boot

Jeffrey Allen Anderson

Spring Web Essentials

Building modern web services with Spring Boot

Jeffrey Allen Anderson

This book is for sale at http://leanpub.com/springwebessentials

This version was published on 2020-03-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 Jeffrey Allen Anderson

Contents

Introduction

Foreword

My first website was online help documentation for the policy administration software used by our call center associates. It was the mid-1990's, and our old help system was clunky. Authoring documents was a multistep process and, with the release of each new version, we had to install it on hundreds of workstations.

By contrast, our new Hypertext Markup Language (HTML) version was easy to maintain, and, being served from a central location, could be changed anytime. Thanks to the cross-platform nature of the Mosaic web browser[1], the content looked the same on Windows, Unix, Mac, and Solaris.

Our magical new HTML user guide was well received, and it wasn't long before the call center manager asked if we could use this technology to allow customers to view account information and make changes over the internet rather than by phone.

Technology choices were limited back then. Our first user-facing website, written in C++ with dynamic content generated through the Common Gateway Interface (CGI) of the webserver, got the job done. It wasn't speedy, and, with our code running directly in the web server, straightforward coding errors could bring down the entire website.

When Java emerged onto the scene, it brought with it the ability to write code once and run it anywhere thanks to the Java Virtual Machine. It also ushered in the era of the application server, creating an actual "three-tier" architecture with distinct layers for presentation, application, and data. Our website performance was much better because we could horizontally scale the web and application tiers independently.

As our website grew in size and complexity, it became clear we needed better ways to manage quality through automated testing. Spring's inversion of control container was a novel approach to wiring up application dependencies in a way that made it easy to test how objects respond to different external behavior, both expected and unexpected.

Spring became a key enabler of automated testing supporting continuous integration (CI). Over time it grew into a complete framework aimed at creating simple, reusable software built on standard patterns that solved universal problems. It's now a set of over twenty projects constructed on the inversion of control foundational core.

My goal for this book is to prepare you to work on both legacy and modern websites built on Spring using a test-driven approach. The first part covers building application programming interfaces (APIs) that power the "modern" web. In the second part, we build a more traditional web application.

[1] https://en.wikipedia.org/wiki/Mosaic_(web_browser)

Spring has many more projects and capabilities than we can explore in one book. Still, I hope this is an excellent introduction and gives you the confidence to be successful web application developers ready to hit the ground running.

Who should read this book?

If you want to learn how to develop modern web applications with Spring, this book is for you. Part one covers using Spring to power secure, industrial-grade web services ready to support modern web applications built with Hypertext Markup Language (HTML) and JavaScript such as Angular, React, and Vue. In part two, we develop a more traditional server-side HTML web application.

Why this book and not another Spring book on the market?

The first version of Spring came out in 2003 when Java lacked many of the capabilities that make it a great language today, like generics, annotations, and variable arguments. As the language brought more natural ways of doing things, Spring improved to take advantage of them. Strong backward compatibility is one of the five major design philosophies[2] of Spring, but it also means there are new, older, and much older ways of doing things. Documenting all of them makes for thick books.

Second, older books don't cover the magic that is Spring Boot[3], which significantly simplifies configuring and running Spring applications. It's no longer an arduous task to get a Spring application up and running, so why should you learn how to do lots of unnecessary work?

Finally, newer books cover Spring boot but often lack the level of detail needed to deploy real applications to production, including unit tests, bean validation, and security. This book presents a complete example leaving you ready to work on industrial-grade Spring applications.

As a teacher, my reading assignments often had a longer list of what to skip than what to read. I wrote this book to use in the classroom with a focus on current best practices covering what students need to know. We follow a test-driven, standards-based approach. The text is full of links to documentation and reference information for each new concept presented.

What you will need

First, a Mac, Windows, or Linux computer with Java 8 or higher installed. Recently Oracle changed the licensing for Java. For personal use, it's still free to use, but if you plan to use your code in production, consider using an OpenJDK[4] distribution such as the Spring recommended AdoptOpenJDK[5]. If you are running Linux, OpenJDK is available through your distribution's package manager.

Second, an Integrated Development Environment (IDE). For this book, I will be using IntelliJ IDEA[6]

[2] https://docs.spring.io/spring/docs/5.2.3.RELEASE/spring-framework-reference/overview.html#overview-philosophy
[3] https://spring.io/projects/spring-boot
[4] https://en.wikipedia.org/wiki/OpenJDK
[5] https://adoptopenjdk.net/
[6] https://www.jetbrains.com/idea/

from JetBrains. I pay for the Ultimate edition because I want to support the great work they do, but the Community edition is free and will work for everything we do in this book.

Another worthy IDE alternative is Visual Studio Code[7], frequently referred to as "VS Code." It's brought to you by Microsoft and is also free to download and use.

Third, a web services testing tool like Postman[8]. It's free to use, and creating a Postman account is optional.

Finally, you will need a solid working knowledge of Java. I take the time to explain how Spring works and provide lots of code, so don't worry too much if you are still learning.

About the author

For as long as he can remember, Jeff has been fascinated by gadgets. His all-time favorite Christmas gift was a Science Fair 100 in 1 Kit, because, with each project he tried, he learned more about how electronic components work together to make something useful.

The knowledge he gained building "space-age" electronics in the 1970s set him on his path as a lifelong author, student, teacher, enterprise IT architect, software developer, and data geek. He is an Amazon Web Services (AWS) Academy Accredited Instructor and holds three AWS certifications: Cloud Practitioner, Solutions Architect Associate, and Security Specialty. When he logs out, he loves motorcycle riding, camping, traveling, and racing.

His career spans four decades in many roles, including developer, lead developer, application architect, infrastructure architect, data architect, and, most recently, educator. In the early days, his favorite programming language was Pascal and, most recently, Go[9]. He will always have a soft spot in his heart for Java. He wrote his first Java program in 1997 and spent the next fifteen years using it to develop websites, web services, and other software for a Fortune 100 company. Now, he teaches Java for Columbus State Community College.

Acknowledgements

Cover art photo by Vural Yavas from Pexels

[7] https://code.visualstudio.com/
[8] https://www.postman.com/
[9] https://golang.org/

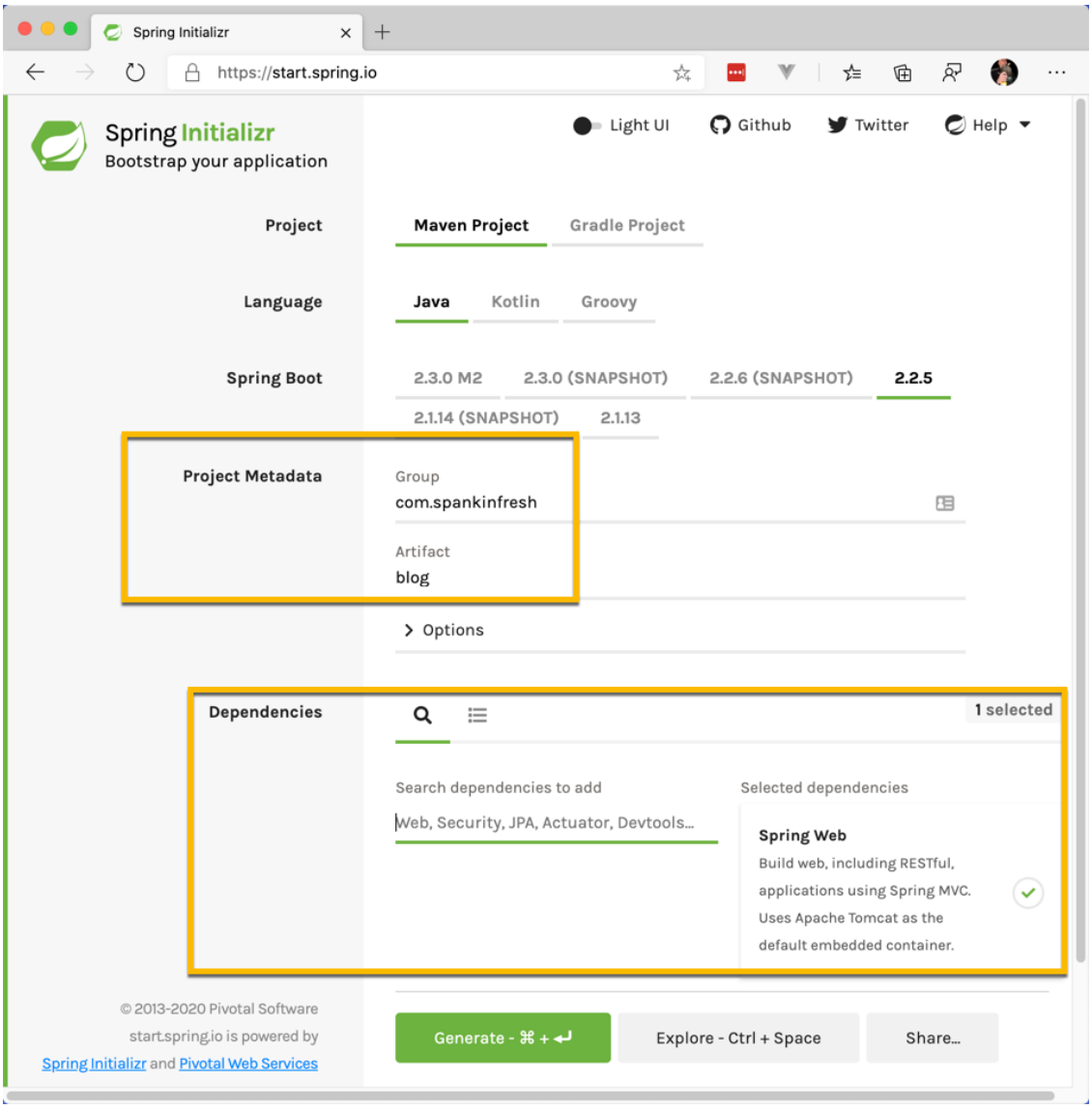
Part One: Web Services For The Modern Age

In part one, we will develop a set of Representational State Transfer (REST) web services that power the fictitious Spankin' Fresh Farmers Market food blog. You can see the Application Programming Interface (API) specification for what we are building on SwaggerHub[10], and we will secure them using a standards based approach[11].

[10] https://app.swaggerhub.com/apis/DataDaddy/spring-web_essentials_blog_api/
[11] https://www.okta.com/identity-101/whats-the-difference-between-oauth-openid-connect-and-saml/

Getting started

Create a new Maven project using Spring Initializr



Spring Initializr web page

1. Open your browser and go to https://start.spring.io/[12]

[12] https://start.spring.io/

2. Keep the defaults of "Maven Project" for the project, "Java" for language, and the automatically chosen Spring Boot version.
3. Under Project Metadata, enter:

- `com.spankinfresh` for the "Group" — **note**: if you are not familiar with Apache Maven[13], in a production setting, this should globally identify your organization in a Maven repository. Since we won't be publishing our app to one, what you enter here is less important. Use any valid domain name you own or use `com.spankinfresh` so your code stays consistent with the content in this book.
- `blog` for the "Artifact" — **note**: the value should uniquely identify your app within the group.

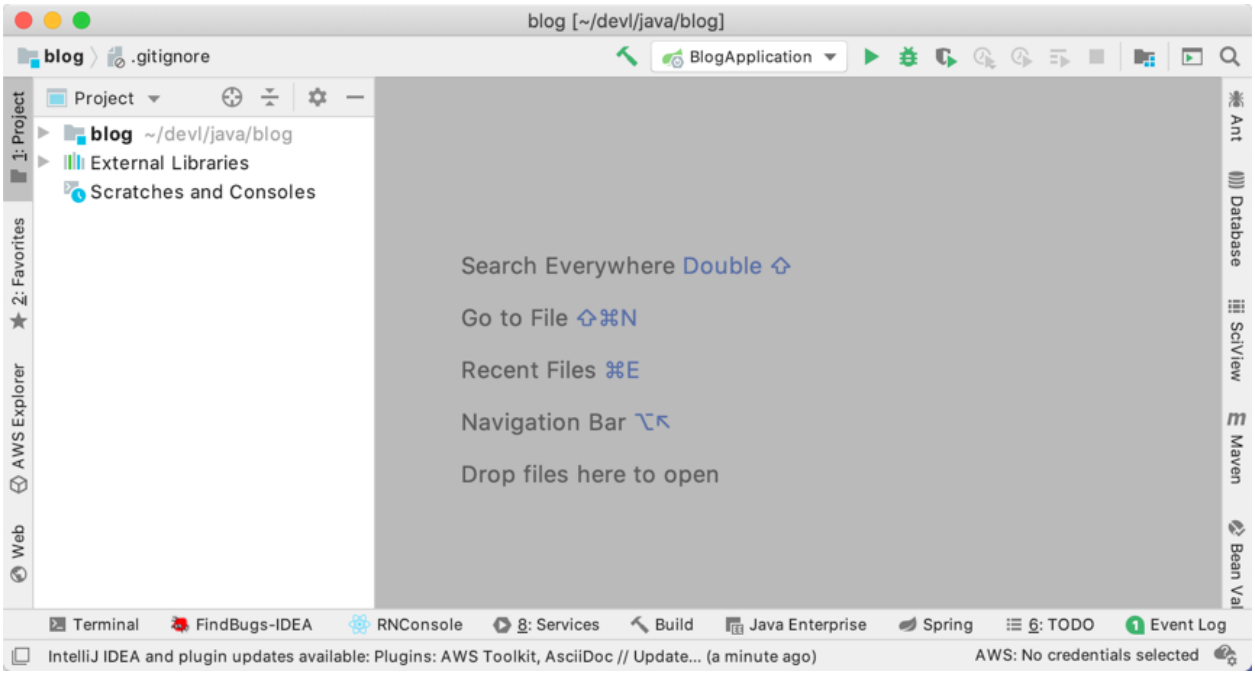
4. In the dependencies search field, enter "web" then click on "Spring Web" from the search results that appear, making sure it moves under "Selected dependencies."
5. Finally, choose "Generate," and Spring Initializr will download a `blog.zip` file with your starter project.

Extract and open the project

1. Unzip the file you downloaded into a new folder under your development directory.
2. Start IntelliJ.
3. Close any projects that come up until you see the "Welcome" screen
4. From the welcome screen, choose "Open" then navigate to the top-level directory of the extracted `blog.zip` archive from step 1, above.

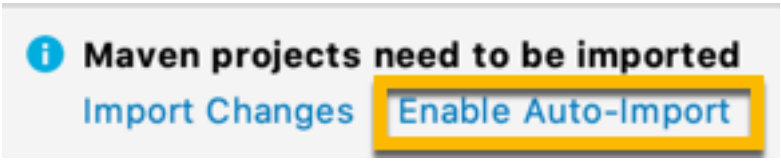
It may take a while for IntelliJ to import the project and download all dependencies, so patience is a virtue. Within a few minutes, your workspace should look like this:

[13] http://maven.apache.org/



IntelliJ IDEA workspace after opening our new project

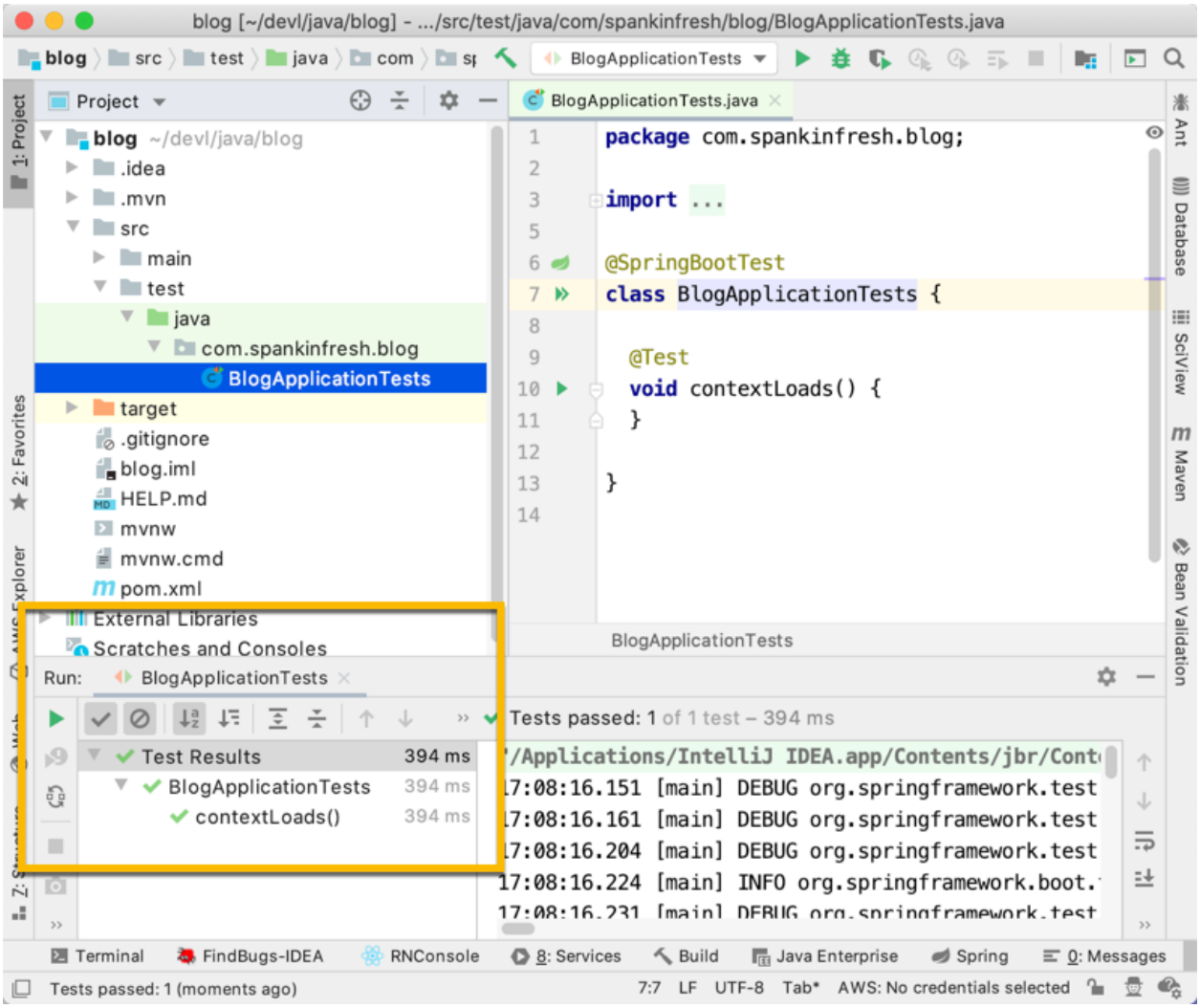
NOTE: You may see the message, "Maven projects need to be imported" at the bottom-right portion of the workspace. If so, choose "Enable Auto-Import". Doing so means each time you change the `pom.xml` file, Maven will automatically download new dependencies.



Import Maven projects message

Make sure you can run your new application

When you created your project, Spring Initializr included everything you need to run JUnit tests. From the project pane, expand the directories under `blog` › `src` › `test` › `java/-com.spankinfresh.blog`



Running your first test

Right-click on `BlogApplicationTests` then choose **Run BlogApplicationTests**. If everything is working correctly, you should see green check marks next to "contextLoads()" in the run tests panel

Congratulations! In just a few minutes, you built a Spring application.

A closer look at the Maven Project Object Model (POM) file

Since we chose a maven project, the Spring Initializr created a Maven POM file[14] for us. The contents, shown below, tell Maven about our dependencies and how to build our project.

[14] http://maven.apache.org/pom.html

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.4.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.spankinfresh</groupId>
  <artifactId>blog</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>blog</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>8</java.version>
  </properties>

  <dependencies>
    <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>
      <exclusions>
        <exclusion>
          <groupId>org.junit.vintage</groupId>
          <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>

  <build>
    <plugins>
```

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

</project>
```

Looking in the file, you will see the entries we made in the Spring Initializr screen for "group" and "artifact" included just after the `<parent>...</parent>` section. The version defaults to `0.0.1-SNAPSHOT`. The combination of these three is known as the Maven coordinates[15], which denote a specific version of the project. The name[16] defaults to the artifact id, but you can change it to something meaningful like a code or marketing name for the application.

Spring boot provides many project starters[17] which take advantage of the inheritance[18] feature of Maven, to provide a base configuration and declare a minimum set of dependencies. The `<parent>...</parent>` section of our POM points to the `spring-boot-starter-parent` artifact from the `org.springframework.boot` group. The version will correspond to the value selected on the Spring Initializr start page.

The properties[19] section allows us to define key-value pairs and access the values by name anywhere in the file. In our case, we have a `java.version` property which we can reference as `${java.version}` wherever we would otherwise supply the Java version.

As you recall, we chose "Spring Web" as the only dependency on the Spring Initializr start page. It's listed first in the dependencies[20] section and provides everything we need to build web applications, including REST web services using Spring model-view-controller (MVC). It uses Apache Tomcat[21] as the default embedded container. Before Spring Boot, we had to manually install and configure Tomcat before we could run our web applications.

Because every well-written application has lots of tests, `spring-boot-starter-test` is automatically included with a "test" scope, meaning it's dependencies will be added when building and running unit tests but not when running the application. This starter includes several dependencies for testing Spring Boot applications with libraries, including JUnit[22], Hamcrest[23] and Mockito[24]. The exclusion[25] for `junit-vintage-engine` tells Maven we only plan to write JUnit 5 tests. If we wanted to write JUnit 4 tests, we would remove that exclusion. See testing[26] for more.

[15] http://maven.apache.org/pom.html#Maven_Coordinates
[16] http://maven.apache.org/pom.html#More_Project_Information
[17] https://docs.spring.io/spring-boot/docs/2.2.4.RELEASE/reference/html/using-spring-boot.html#using-boot-starter
[18] http://maven.apache.org/pom.html#Inheritance
[19] http://maven.apache.org/pom.html#Properties
[20] http://maven.apache.org/pom.html#Dependencies
[21] https://tomcat.apache.org/
[22] https://junit.org/junit5/
[23] http://hamcrest.org/JavaHamcrest/
[24] https://site.mockito.org/
[25] http://maven.apache.org/pom.html#Exclusions
[26] https://docs.spring.io/spring-boot/docs/2.2.4.RELEASE/reference/html/spring-boot-features.html#boot-features-testing

Finally, the plugins[27] section of the POM file has the Spring Boot Maven plugin[28]. When we build our project, Maven runs it to package our project as an executable Java Archive (JAR) file.

You may have noticed Spring Initializr also provided `mvnw` and `mvnw.cmd` in the top-level directory of our project. The former is a shell script for use on Linux, Unix, or Mac. The latter, a batch file for Windows. If you are using IntelliJ, you won't need them but, they come in handy if you want to package your project from the command line. Don't worry if you have not installed Maven. The scripts will download and install it for you.

A closer look at the main Java application class

Spring Initializr created two Java classes for us. The main application and a unit test. Let's look at the main application class first.

```
1  package com.spankinfresh.blog;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6  @SpringBootApplication
7  public class BlogApplication {
8
9    public static void main(String[] args) {
10     SpringApplication.run(BlogApplication.class, args);
11   }
12
13 }
```

The executable JAR file Maven produces automatically calls the `main` method in the class above, which, in turn, calls SpringApplication.run[29] to bootstrap and launch the Spring application. Decorating a class with @SpringBootApplication[30] is a more succinct way of decorating it with @Configuration[31], @EnableAutoConfiguration[32], and @ComponentScan[33].

`@ComponentScan` tells Spring to scan the base package, `com.spankinfresh.blog` in our case, and all sub-packages for classes annotated with `@Configuration` which it uses to configure the application. Component scanning works with annotation-based container configuration[34] to simplify and

[27] http://maven.apache.org/pom.html#Plugins
[28] https://docs.spring.io/spring-boot/docs/2.2.4.RELEASE/reference/html/using-spring-boot.html#using-boot-maven-plugin
[29] https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/SpringApplication.html#run-java.lang.Class-java.lang.String...-
[30] https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/SpringBootApplication.html
[31] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html
[32] https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/EnableAutoConfiguration.html
[33] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html
[34] https://docs.spring.io/spring/docs/5.2.3.RELEASE/spring-framework-reference/core.html#beans-annotation-config

automate the process. It also follows the DRY (don't repeat yourself) principle, which is a problem with the original XML-based container configuration[35].

`@EnableAutoConfiguration` takes it a step further by automatically applying an additional configuration based on what it finds in the classpath. Even though we did not provide any for a web application, since we included `spring-boot-starter-web` in our POM file, Spring assumes we want a web server and automatically configures it for us. This powerful feature is one of the reasons we have a fully functioning web application with one annotation and just a few lines of code.

A closer look at the JUnit test class

```
1  package com.spankinfresh.blog;
2
3  import org.junit.jupiter.api.Test;
4  import org.springframework.boot.test.context.SpringBootTest;
5
6  @SpringBootTest
7  class BlogApplicationTests {
8
9    @Test
10   void contextLoads() {
11   }
12
13 }
```

Our application doesn't yet do anything, but we want to make sure Spring can run it. By decorating a test with `@SpringBootTest`, Spring will search for a class annotated with `@SpringBootApplication` and use it to configure the application.

The test method doesn't explicitly test anything, but its mere presence causes Spring to load the application. The test will pass if everything is working or throw an error if not, causing the test to fail.

Git committed!

More times than I can count, I have messed something up and had no idea why it wasn't working. It's tremendously helpful to be able to revert to a known working state or simply compare my changes with my last commit. For this reason, I highly recommend using version control and committing often.

[35] https://docs.spring.io/spring/docs/5.2.3.RELEASE/spring-framework-reference/core.html#beans-factory-metadata

Create a local Git repository:

1. From the IntelliJ menu, choose VCS > Import into Version Control > Create Git Repository.
2. Choose the automatically selected default of your project's top directory for your local repo.

Check-in your initial commit

1. Add a line with `.mvn` at the bottom of your `.gitignore` file.
2. Choose VCS > Commit.
3. Check the box to the left of "Unversioned Files."
4. Enter "Initial commit" into the "Commit Message" box
5. Press "Commit"

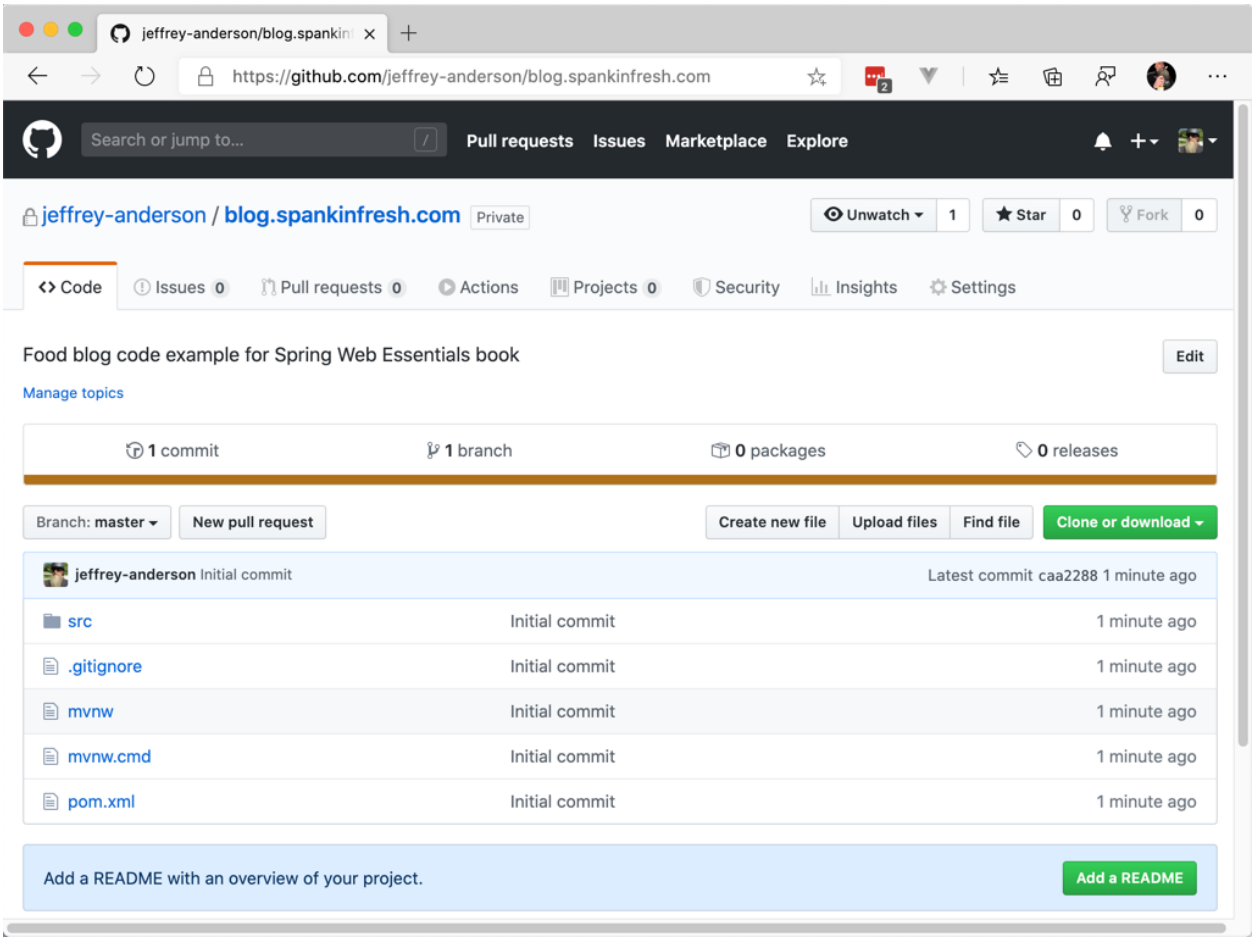
Create a private repo for your project on GitHub

While it might not happen often, hard drives crash, and operating systems become unbootable. I push my changes to GitHub at least daily, so I have a backup of my work.

1. Create a public or private GitHub repository using these instructions[36].
2. Open a terminal window and navigate to your top-level project directory.
3. Copy and paste the "…or push an existing repository from the command line" commands shown on your new repository code tab into the terminal window.

After the push completes, refresh the GitHub browser window and make sure you see something like the view below:

[36] https://help.github.com/en/github/creating-cloning-and-archiving-repositories/creating-a-new-repository



Github view of our new project

Accomplishments

I like to end each chapter with a summary of what we have accomplished. In this chapter, we used Spring Initializr to configure a minimal Spring Boot web application. After reviewing the application configuration and structure, we ran a unit test to make sure the application will run correctly.

Some context

Before we jump right into developing web services, it's good to establish some context. This chapter presents essential concepts that will come in handy as we start writing code.

Web services

In general, a web service is a service offered by a provider and is accessible to clients through the world wide web. The old school term "world wide web" is synonymous with the internet, but web services also run on private networks, commonly referred to as intranets.

Initially, services communicated through context-specific protocols such as the File Transfer Protocol (FTP)[37], formalized in 1971, and Simple Mail Transfer Protocol (SMTP)[38], formalized in 1981. The first version of the Hypertext Transfer Protocol (HTTP), 0.9[39], was introduced in 1991. It was rudimentary with only had one method, GET. In 1995, the Internet Engineering Task Force (IETF) issued Request For Comments (RFC) 1945[40], which defines a complete version 1.0 of the HTTP protocol.

I wrote my first web services using Common Object Request Broker (CORBA)[41]. It allowed us to scale processing to other servers horizontally, but, being designed for programming language and operating system independence, was complex. It came with an interface definition language used to generate platform-specific code for the client and server. Even worse, it used yet another proprietary protocol known as the general Inter-ORB protocol[42].

As my boss likes to say, *the next turn of the crank* was Simple Object Access Protocol (SOAP)[43]. Like CORBA, it is operating system and language agnostic. Messages between components are encoded in Extensible Markup Language (XML)[44] and exchanged via SMTP or, more commonly, HTTP.

It was a distinct improvement on CORBA, but still complicated to use. The verbose nature of XML makes it slow to parse. We use tools to convert Web Services Description Language (WSDL)[45] into Java. Changes in the service often require regeneration of the client and server code. Indeed it's lots of ceremony with a dash of smoke and mirrors thrown in for good measure.

In many respects, SOAP services ushered in a web service golden age. Vendors like IBM were eager to extol the virtues of a service-oriented architecture, also known as "SOA," where services, defined by

[37] https://tools.ietf.org/html/rfc114
[38] https://tools.ietf.org/html/rfc788
[39] https://www.w3.org/Protocols/HTTP/AsImplemented.html
[40] https://tools.ietf.org/html/rfc1945
[41] https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture
[42] https://en.wikipedia.org/wiki/General_Inter-ORB_Protocol
[43] https://en.wikipedia.org/wiki/SOAP
[44] https://en.wikipedia.org/wiki/XML
[45] https://en.wikipedia.org/wiki/Web_Services_Description_Language

WSDL could advertise and interact on an enterprise service bus (ESB)[46]. To many of us, it seemed like marketing hype to sell proprietary products with fat maintenance contracts. Still, it did get everyone thinking about how systems could collaborate in a decoupled fashion.

Representational State Transfer (REST)

While everyone was busy rewriting their CORBA services using SOAP, Roy Fielding was completing his Doctoral Dissertation, Architectural Styles and the Design of Network-based Software Architectures[47]. This paper, published in 2000, was the pivotal moment when REST was born.

Unlike its predecessors, it's not a standard issued by a governing body. Instead, it's an architectural style that describes how to build web services with messages in any language, format, or encoding, exchanged using standard HTTP.

> The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g., a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.[48]

Resources can be represented in a variety of ways:

> REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A representation is a sequence of bytes, plus representation metadata to describe those bytes. Other commonly used but less precise names for a representation include: document, file, and HTTP message entity, instance, or variant.[49]

Resources are identified by a Uniform Resource Identifier (URI), which is a generalization of the more familiar term Uniform Resource Locator (URL). Wikipedia has an excellent overview of the topic[50], and RFC 3986[51] has the official specification.

Clients use HTTP request methods[52] as verbs on what action to take. The server responds with an HTTP status code[53] that gives the result of the request attempt.

[46] https://www.ibm.com/cloud/learn/esb
[47] https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
[48] Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000, § 5.2.1.1 Resources and Resource Identifiers
[49] Ibid, § 5.2.1.2 Representations
[50] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
[51] https://tools.ietf.org/html/rfc3986
[52] https://tools.ietf.org/html/rfc7231#section-4
[53] https://tools.ietf.org/html/rfc7231#section-6

REST uses media types[54] to denote the format of a resource representation. For example, the same image can be represented in Portable Network Graphics (PNG) format and Joint Photographic Experts Group (JPEG) format. Thanks to the early popularity of SOAP, XML was the original format of choice for text, but, as JavaScript gained predominance in web applications, JavaScript Object Notation (JSON) is now the most common standard textual representation.

Since a given resource may be represented in any number of ways, section 3.4[55] of RFC 7231 describes how clients and servers agree on a mutually acceptable format, language, or encoding.

Another feature of REST is that all operations are stateless.

> All REST interactions are stateless. That is, each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it. This restriction accomplishes four functions: 1) it removes any need for the connectors to retain application state between requests, thus reducing consumption of physical resources and improving scalability; 2) it allows interactions to be processed in parallel without requiring that the processing mechanism understand the interaction semantics; 3) it allows an intermediary to view and understand a request in isolation, which may be necessary when services are dynamically rearranged; and, 4) it forces all of the information that might factor into the reusability of a cached response to be present in each request.[56]

Since REST uses HTTP for it's the transport layer, it also allows for caching:

> A cache is able to determine the cacheability of a response because the interface is generic rather than specific to each resource. By default, the response to a retrieval request is cacheable and the responses to other requests are non-cacheable. If some form of user authentication is part of the request, or if the response indicates that it should not be shared, then the response is only cacheable by a non-shared cache. A component can override these defaults by including control data that marks the interaction as cacheable, non-cacheable or cacheable for only a limited time.[57]

More about HTTP

A full review of HTTP is outside the scope of this book but the main set of standards are listed below:

- RFC 7230[58]: Message Syntax and Routing
- RFC 7231[59]: Semantics and Content

[54] https://tools.ietf.org/html/rfc6838
[55] https://tools.ietf.org/html/rfc7231#section-3.4
[56] Ibid, § 5.2.2 Connectors
[57] Ibid, § 5.2.2 Connectors
[58] https://tools.ietf.org/html/rfc7230
[59] https://tools.ietf.org/html/rfc7231

- RFC 7232[60]: Conditional Requests
- RFC 7233[61]: Range Requests
- RFC 7234[62]: Caching
- RFC 7235[63]: Authentication

Since REST uses HTTP as the transport layer, it's especially important to read RFC 7231, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* so you understand how to interact with resources and properly handle various conditions and responses.

Common HTTP methods

While not an exhaustive list, the methods below are the most used:[64]

Method	Description	Section
GET	Transfer a current representation of the target resource	4.3.1[65]
HEAD	Same as GET, but only transfer the status line and header section	4.3.2[66]
POST	Perform resource-specific processing on the request payload	4.3.3[67]
PUT	Replace all current representations of the target resource with the request payload	4.3.4[68]
DELETE	Remove all current representations of the target resource	4.3.5[69]
OPTIONS	Describe the communication options for the target resource	4.3.7[70]

Common HTTP status codes

The HTTP protocol defines many more status codes, but those listed below are some of the most common:[71]

[60] https://tools.ietf.org/html/rfc7232
[61] https://tools.ietf.org/html/rfc7233
[62] https://tools.ietf.org/html/rfc7234
[63] https://tools.ietf.org/html/rfc7235
[64] RFC 7231, Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content § 4.1. Overview
[65] https://tools.ietf.org/html/rfc7231#section-4.3.1
[66] https://tools.ietf.org/html/rfc7231#section-4.3.2
[67] https://tools.ietf.org/html/rfc7231#section-4.3.3
[68] https://tools.ietf.org/html/rfc7231#section-4.3.4
[69] https://tools.ietf.org/html/rfc7231#section-4.3.5
[70] https://tools.ietf.org/html/rfc7231#section-4.3.7
[71] Ibid § 6.1. Overview of Status Codes

Code	Description	Defined in...
200	OK	Section 6.3.1[72]
201	Created	Section 6.3.2[73]
204	No Content	Section 6.3.5[74]
301	Moved Permanently	Section 6.4.2[75]
302	Found	Section 6.4.3[76]
303	See Other	Section 6.4.4[77]
307	Temporary Redirect	Section 6.4.7[78]
400	Bad Request	Section 6.5.1[79]
401	Unauthorized	Section 3.1 of RFC7235[80]
403	Forbidden	Section 6.5.3[81]
404	Not Found	Section 6.5.4[82]
405	Method Not Allowed	Section 6.5.5[83]
415	Unsupported Media Type	Section 6.5.13[84]
500	Internal Server Error	Section 6.6.1[85]

The protocol defines the mechanics of the conversation between the client and the server. You are free to design your application in a way that makes sense, provided you are operating within these guidelines

For example, if the same content is sent via POST multiple times, you can return a 201 created response and store a new copy with each invocation. Alternatively, you can do that the first time then, for each subsequent invocation, send a 303 see other response with the location of the original resource. Similarly, if the client sends a PUT with an ID in the URL that doesn't exist, you are free to create it as if a POST was sent or reject it with a 404 not found error.

The key here is to be consistent and document your APIs using the OpenAPI Specification[86] or another standard format.

[72] https://tools.ietf.org/html/rfc7231#section-6.3.1
[73] https://tools.ietf.org/html/rfc7231#section-6.3.2
[74] https://tools.ietf.org/html/rfc7231#section-6.3.5
[75] https://tools.ietf.org/html/rfc7231#section-6.4.2
[76] https://tools.ietf.org/html/rfc7231#section-6.4.3
[77] https://tools.ietf.org/html/rfc7231#section-6.4.4
[78] https://tools.ietf.org/html/rfc7231#section-6.4.7
[79] https://tools.ietf.org/html/rfc7231#section-6.5.1
[80] https://tools.ietf.org/html/rfc7235#section-3.1
[81] https://tools.ietf.org/html/rfc7231#section-6.5.3
[82] https://tools.ietf.org/html/rfc7231#section-6.5.4
[83] https://tools.ietf.org/html/rfc7231#section-6.5.5
[84] https://tools.ietf.org/html/rfc7231#section-6.5.13
[85] https://tools.ietf.org/html/rfc7231#section-6.6.1
[86] https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md#openapi-specification

Our first application programming interface (API)

Now that we understand key concepts of web services let's get back to the fun! Over the next few chapters, we will build a "CRUD" (create, read, update, and delete) API for postings in our food blog using a test-driven development approach.

The OpenAPI Specification

Throughout this book, I will express API requirements using the OpenAPI specification[87]. Quoting from their documentation:[88]

> The OpenAPI Specification is a community-driven open specification within the OpenAPI Initiative, a Linux Foundation Collaborative Project.
>
> The OpenAPI Specification (OAS) defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interface descriptions have done for lower-level programming, the OpenAPI Specification removes guesswork in calling a service.

After we complete this chapter, our API will match this specification:

```
openapi: 3.0.1
info:
  title: "spring-web-essentials-blog-api"
  version: "0.1.0"
  description:
    Initial specification for the food blog example Application
    Programming Interface (API) in the Spring Web Development
    Essentials book.

```

[87] https://github.com/OAI/OpenAPI-Specification/
[88] The OpenAPI Specification

```
paths:
  /api/articles:
    post:
      summary: Post a new blog article
      responses:
        '201':
          description: The blog post was created successfully
          headers:
            Location:
              description: The location of the newly created blog post
              schema:
                type: string
          content: {}
```

From this spec above, our API supports one method, `POST`, at path `/api/articles`. It always returns a 201, created status with one header, `Location`, which has a valid URL for the newly created blog post. Finally, our method returns no content.

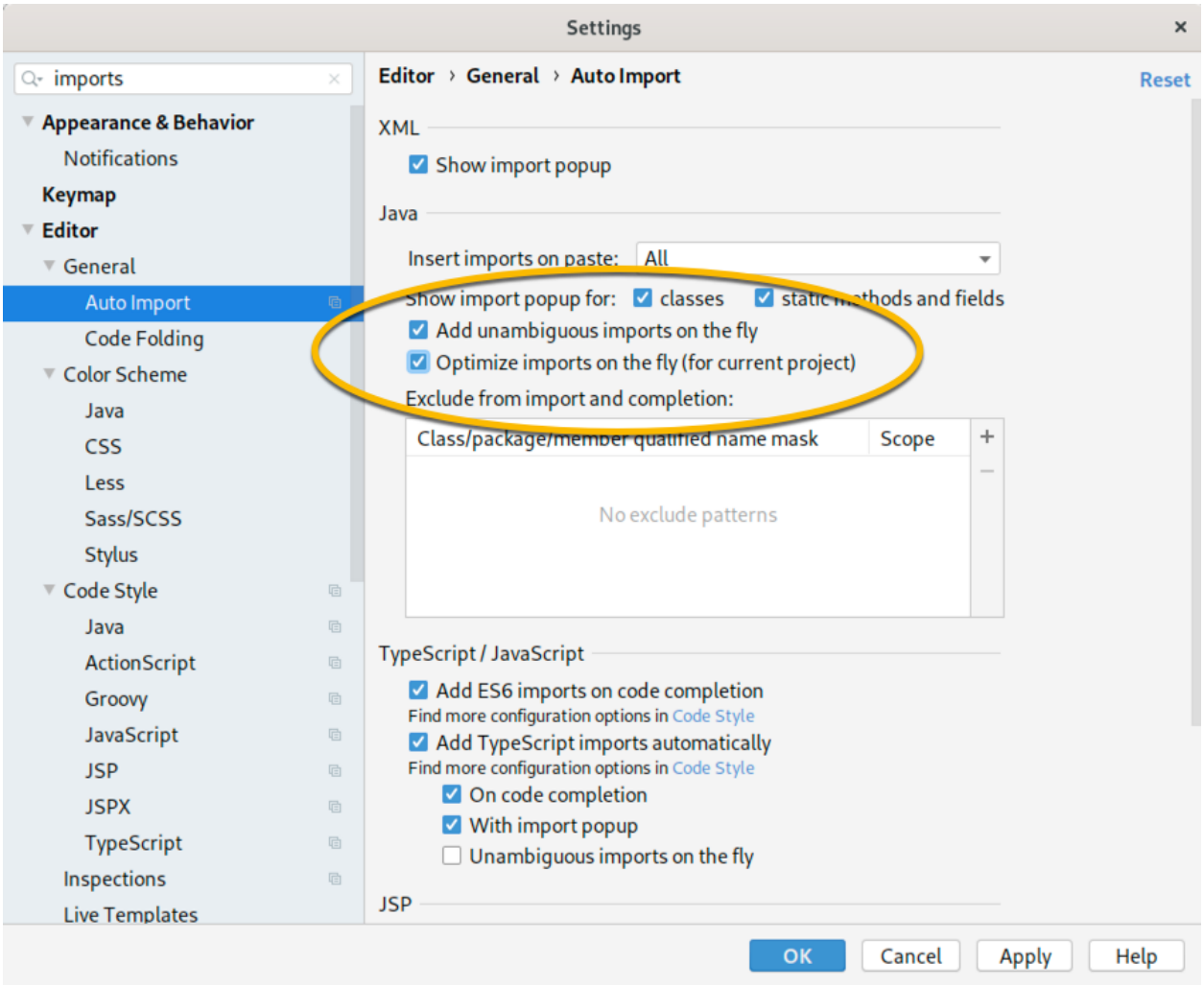
You can view a more interactive version on SwaggerHub[89]

Quick tip before we get started

Importing classes is a real pain. The best way to spend less time doing so is to have IntelliJ do it automatically where there is only one possible match. I also like to have imports optimized on the fly, so I don't have to worry about unused imports.

To get this same behavior on your machine, open the settings dialog, and check the options shown below, then click "OK."

[89] https://app.swaggerhub.com/apis/DataDaddy/spring-web_essentials_blog_api/0.1.0



Enabling auto-add imports in project settings

Create a new blog posting

From the RFC 7231 specification:[90]

> If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server SHOULD send a 201 (Created) response containing a Location header field that provides an identifier for the primary resource created (Section 7.1.2) and a representation that describes the status of the request while referring to the new resource(s).

I like to start simple and iteratively add complexity. Doing things one step at a time means when something stops working, I only have one thing to debug. Unit tests help me get going with the least amount of work. In this case, the spec calls for us to return:

[90] RFC 7231, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* § 4.3.3 POST

1. a 201 created status
2. a `Location` header with the URL of the newly created resource

Create blog posting test

1. In the Project pane, under `src/test/java`, right-click on `com.spankinfresh.blog`
2. Choose **New > Java Class**
3. Enter `api.BlogPostControllerTests` as the class name
4. Choose "Add" to add our new class to version control
5. Decorate the class with `@SpringBootTest` and `@AutoConfigureMockMvc`
6. Add a RESOURCE_URI constant with a value of "/api/articles"
7. Add the test method shown below

Your class should look like this:

```
1  package com.spankinfresh.blog.api;
2
3  import org.junit.jupiter.api.DisplayName;
4  import org.junit.jupiter.api.Test;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
7  import org.springframework.boot.test.context.SpringBootTest;
8  import org.springframework.test.web.servlet.MockMvc;
9  import static
10   org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
11 import static
12   org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
13
14 @SpringBootTest
15 @AutoConfigureMockMvc
16 public class BlogPostControllerTests {
17
18   private static final String RESOURCE_URI = "/api/articles";
19
20   @Test
21   @DisplayName("T01 - Post returns status of CREATED")
22   public void test01(@Autowired MockMvc mockMvc)
23     throws Exception {
24       mockMvc.perform(post(RESOURCE_URI))
25         .andExpect(status().isCreated());
26   }
27 }
```

Our test interacts with a web server, but Spring doesn't start one by default with `@SpringBootTests`. Annotating the test class with `@AutoConfigureMockMvc` provides a mock web server that we pass to each test method by way of the `MockMvc` parameter allowing us to perform operations against the API under test.

In the example above, we perform a POST to our resource URI and expect it returns a status of created. Section 25.3.5. Testing with a mock environment[91] of the Spring Boot reference has additional details and examples.

The MocMVC parameter is annotated with @Autowired[92] which tells Spring to locate a bean of a matching type and automatically provide an instance of it. You can read more about how this works in the Spring documentation, section 1.9.2. Using @Autowired[93].

`@DisplayName` is new with JUnit 5. Without this annotation, the method name shows up in the test output. Good programmers used verbose method names to convey the purpose of the test. Now, we can simply write it clearly in the annotation, making it much easier to read. I typically number the methods and include them in the display text for a faster association.

Run the test and verify the result

1. From the project pane, right click `BlogPostControllerTests`
2. Choose "Run BlogPostControllerTests"
3. Verify the test fails with the following error:

```
java.lang.AssertionError: Status expected:<201> but was:<404>
Expected :201
Actual   :404
```

Note you might have to scroll through the test output to find the text above.

Get our test to pass

It should come as no surprise that the server returned a 404 not found error. We haven't created a controller and method to handle the request, so let's fix that now.

1. In the Project pane, under `src/main/java`, right-click on `com.spankinfresh.blog`
2. Choose **New > Java Class**
3. Enter `api.BlogPostController` as the class name
4. Choose "Add" to add our new class to version control
5. Decorate the class with `@RestController` and `@RequestMapping("/api/articles")`
6. Add the method shown below, decorated with `@PostMapping`

Your class should look like this:

[91] https://docs.spring.io/spring-boot/docs/2.2.4.RELEASE/reference/html/spring-boot-features.html#boot-features-testing-spring-boot-applications-testing-with-mock-environment
[92] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html
[93] https://docs.spring.io/spring/docs/5.2.3.RELEASE/spring-framework-reference/core.html#beans-autowired-annotation

```
1  package com.spankinfresh.blog.api;
2
3  import org.springframework.http.HttpStatus;
4  import org.springframework.http.ResponseEntity;
5  import org.springframework.web.bind.annotation.PostMapping;
6  import org.springframework.web.bind.annotation.RequestMapping;
7  import org.springframework.web.bind.annotation.RestController;
8
9  @RestController
10 @RequestMapping("/api/articles")
11 public class BlogPostController {
12
13   @PostMapping
14   public ResponseEntity createBlogEntry () {
15     return new ResponseEntity(HttpStatus.CREATED);
16   }
17
18 }
```

Decorating this class with @RestController[94] is another convenience annotation. Instead, we could have used @Controller[95] and @ResponseBody[96]. Either way, the result is the same. More on model-view-controller later but classes with the `@Controller` designation handle web requests. They control both old school server-side web sites, returning a model and view, or, as indicated by the `@ResponseBody` designation, a raw REST API response.

We annotated the class with `@RequestMapping("/api/articles")` so Spring maps web requests for any HTTP method to this controller. Also, since the `/api/articles` URI fragment is on the class level annotation, it is common to all request handlers in the controller. URI fragments or path variables in method mappings are appended to the common prefix.

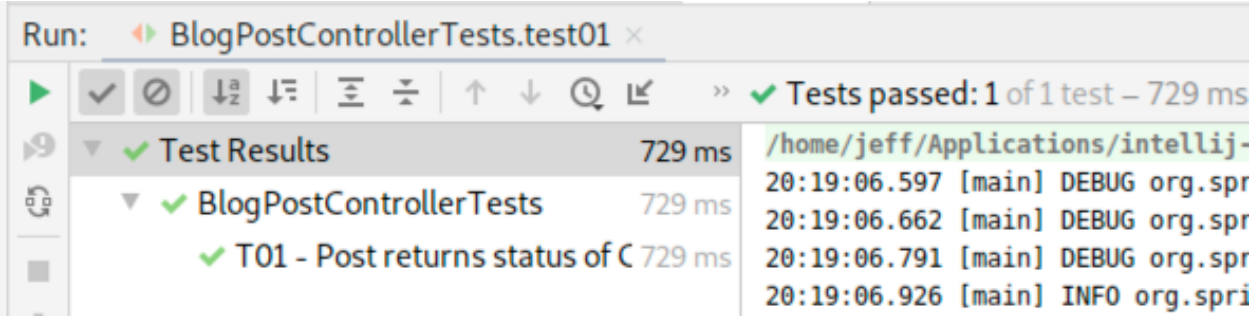
Our createBlogEntry method is decorated with @PostMapping[97] meaning it handles POST requests. As we will see later, there is also @GetMapping[98], @PutMapping[99], @DeleteMapping[100], and @PatchMapping[101] annotations for each of those HTTP methods.

@ResponseEntity[102] extends HttpEntity[103] and adds a HttpStatus[104] code. As such, we can return a full HTTP response, including headers, body, and a status code. Our method above only returns a status code, but that will change in the next section.

[94] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html
[95] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Controller.html
[96] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html
[97] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PostMapping.html
[98] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/GetMapping.html
[99] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PutMapping.html
[100] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/DeleteMapping.html
[101] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PatchMapping.html
[102] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html
[103] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpEntity.html
[104] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpStatus.html

Run the test and verify it now succeeds

1. From the project pane, right-click `BlogPostControllerTests`
2. Choose "Run BlogPostControllerTests"
3. Verify the test succeeds, as shown below:



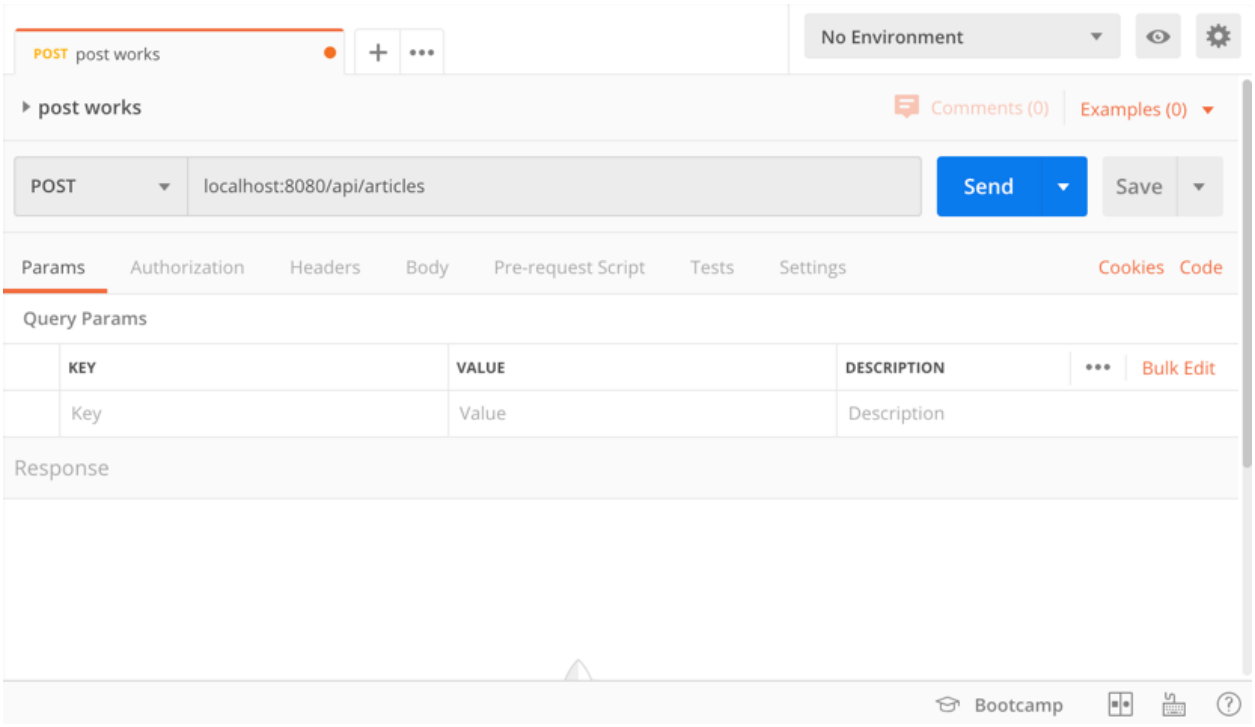
View of our first API test passing

Testing with Postman

Unit testing is fantastic, but did you know we now have a fully functioning REST API application? Let's fire up our app and use Postman[105] to test it.

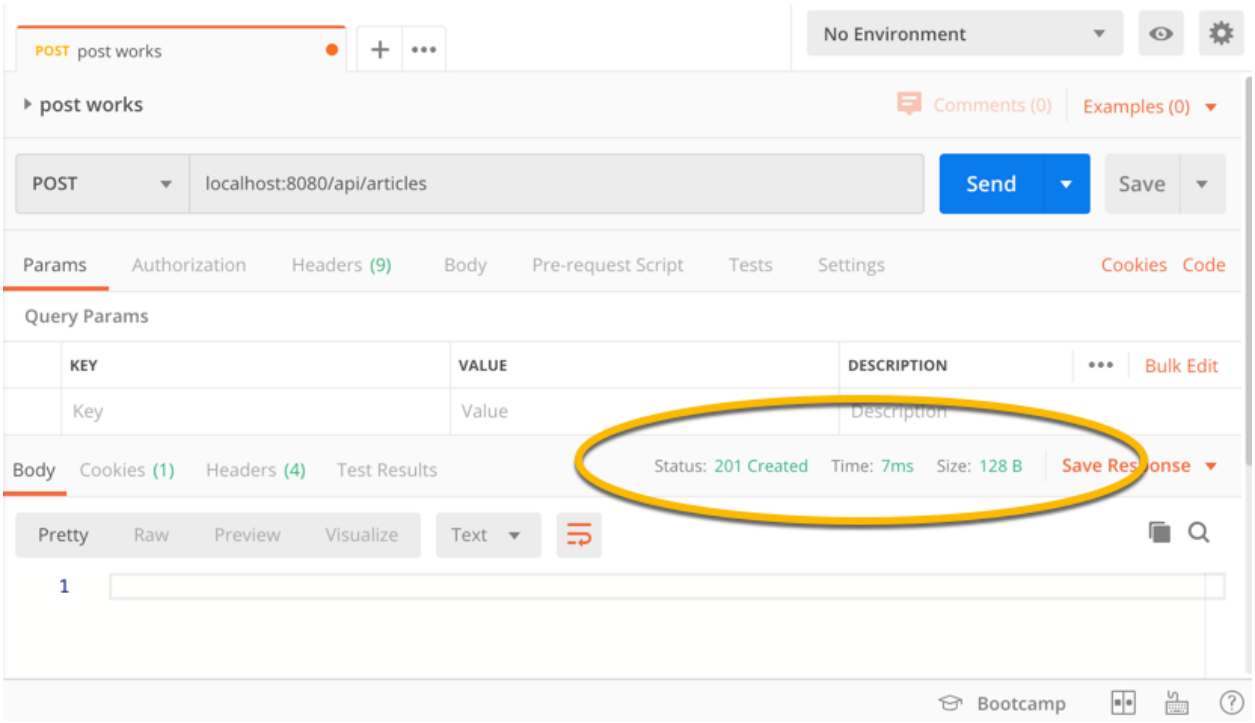
1. From the Project pane, expand `src/main/java/com.spankinfresh.blog` if it's not already open.
2. Right-click on `BlogApplication` and choose "Run BlogApplication". After a few seconds, you should see in the Run pane the message, `Started BlogApplication in x.xxx seconds`.
3. Open Postman. If this is the first time you open the app, it will ask you to create an account. Doing so is optional. If you would rather not, close the dialog that appears or click, "Skip signing in and take me straight to the app" from the bottom center of the splash screen that appears the first time you run the app.
4. If you get the "Create New" dialog, click on "Request" otherwise choose **File > New**... then choose "Request" from the options dialog.
5. Enter `post works` for the request name.
6. Towards the bottom, click "+ Create Collection" and enter `Food Blog` and press the checkmark to the right.
7. Click "Save to Food Blog" to save and open your new request in the main area of the workspace.
8. Change the HTTP method from "GET" to "POST" and, to the right, enter `localhost:8080/api/articles` in the URL address field. It should look like this:

[105] https://www.postman.com/



Postman request prior to submission

9. Press the “Send” button to send the request. After a brief moment, the API call should return and display a status of “201 Created” as shown below:



Postman request after submission

10. Click the "Save" button to save your changes.
11. Finally, back in IntelliJ, choose "Run" from the menu then "Stop 'BlogApplication'".

Note: as you make changes to your application, don't forget to stop and restart it before testing your next set of changes. While it's possible to configure IntelliJ do automatically reload updated classes, I have found it to be somewhat inconsistent and slow to reload.

Return a Location header

Now that we have our controller method returning a 201 created response let's move onto the location header.

Add a test for the location

We could update our initial test, but let create a new one so you can easily compare the two.

```
  @Test
  @DisplayName("T02 - Post returns Location header for new item")
  public void test02(@Autowired MockMvc mockMvc) throws Exception {
    MvcResult result =
      mockMvc.perform(post(RESOURCE_URI)).andReturn();

    MockHttpServletResponse mockResponse = result.getResponse();
    assertEquals("http://localhost/api/articles/1",
      mockResponse.getHeader("Location"));
  }
```

In this new test, we chain a call to `.andReturn()` at the end of the `mockMvc.perform()` statement and store the result in a new `MvcResult` variable. Calling `getResponse()` on the result gives us access to the webserver response, including the headers.

In the last two lines, we use the static method `org.junit.jupiter.api.Assertions.assertEquals` to compare the value of the location header to what we are expecting. Since we are not setting that HTTP header in our controller, the test fails:

```
org.opentest4j.AssertionFailedError:
Expected :http://localhost/api/articles/1
Actual   :null
```

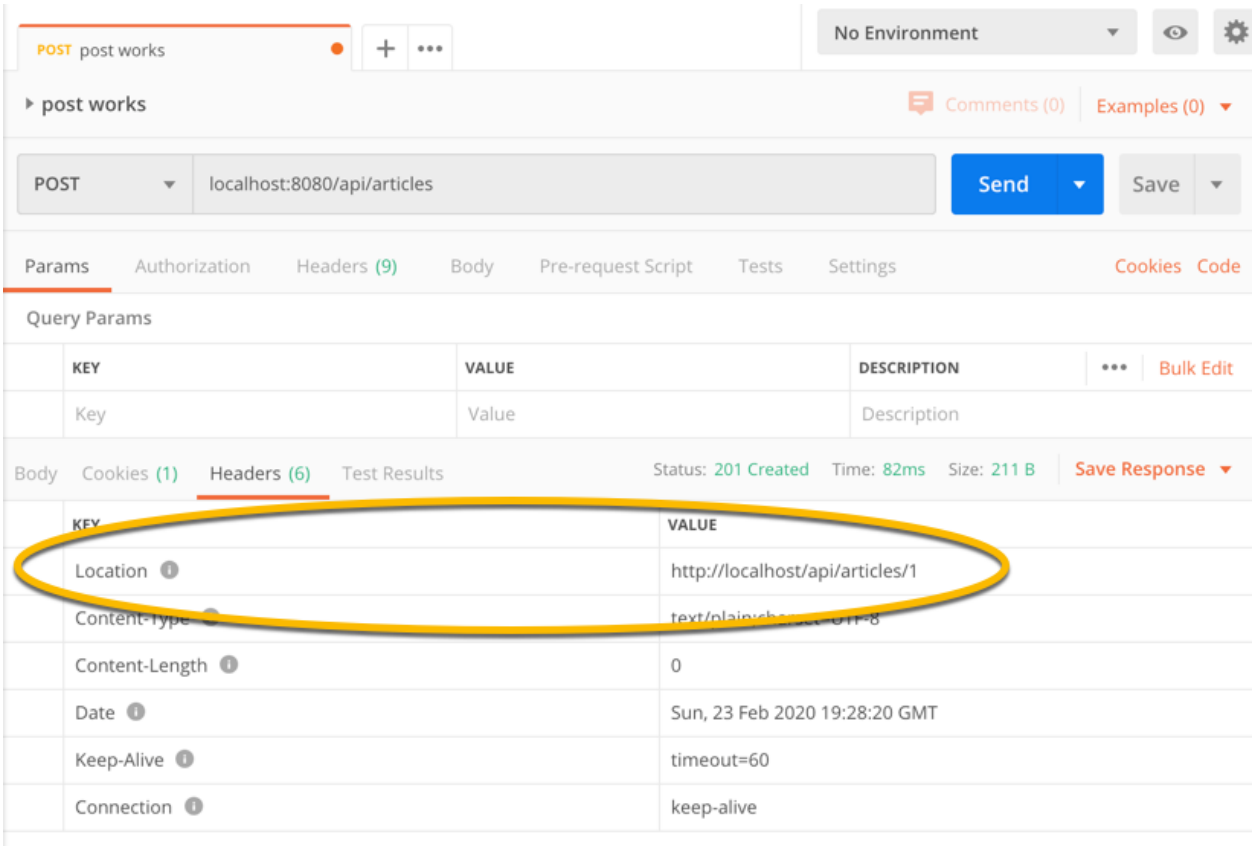
Fix the location header test

Replace the BlogPostController createBlogEntry method with the code shown below:

```
  @PostMapping
  public ResponseEntity createBlogEntry () {
    HttpHeaders headers = new HttpHeaders();
    headers.add("Location", "http://localhost/api/articles/1");
    return new ResponseEntity("", headers, HttpStatus.CREATED);
  }
```

In the version above, we are adding a hard-coded location header and returning it in the ResponseEntity with an empty response body. When you go back and run your blog post controller tests, both should pass.

You can also see the change in Postman. Rerun the `BlogApplication` and resubmit your POST request. The result should be the same, but you will now have five headers and, when you click on the response headers tab, you will see the new Location header:



Postman request after submission with location header

Accomplishments

Yes, I realize we hardcoded a return value to make the test pass and aren't posting, let alone saving anything. To create a posting for our food blog, we need to define what one is and how to represent it. We also need a database to store the articles. Had we included these concerns right from the start, it would have overcomplicated things.

Application development is like constructing a high-rise building. It must be done floor by floor. Layering on too much too quickly increases the risk it will collapse under its weight. Now that the footers are in place and the foundation laid, we are ready to make it fully functional, knowing we have a running application that is ready to handle our REST API requests.

Adding an object model

Now that we have a working API, it's time to add a representation for articles posted to our blog. By creating a domain class, Spring can automatically convert it to and from JavaScript Object Notation (JSON). In this chapter, we add the domain class then update our controller to extract it from the request body and return a copy in the response.

Updated OpenAPI specification

The specification below describes what we will be building in this chapter:

```
openapi: 3.0.1
info:
  title: "spring-web-essentials-blog-api"
  version: "0.2.0"
  description:
    Specification for the food blog example Application
    Programming Interface (API) in the Spring Web Development
    Essentials book. This version includes the object model.
paths:
  /api/articles:
    post:
      summary: Post a new blog article
      requestBody:
        description: New food blog article
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/BlogPost'
      responses:
        '201':
          description:
            The blog post was created successfully and the
            response body has a representation of the newly
            saved article
          headers:
            Location:
```

```
28                description:
29                  The location of the newly created blog post
30                schema:
31                  type: string
32            content:
33              application/json:
34                schema:
35                  $ref: '#/components/schemas/BlogPost'
36  components:
37    schemas:
38      BlogPost:
39        type: object
40        description: Representation of an article posted to our blog
41        properties:
42          id:
43            type: integer
44            description: The article ID
45            default: 0
46          category:
47            type: string
48            description: The category group for this article
49            default: null
50          datePosted:
51            type: string
52            format: date-time
53            description: RFC 3339 formatted date the article was posted
54            default: null
55          title:
56            type: string
57            description: Blog post title
58            default: null
59          content:
60            type: string
61            description: Markdown blog post content
62            default: null
```

The SwaggerHub version is available here[106]. It has a new schema for `BlogPost`, which is passed as `application/json` in the `requestBody` to the `post` method. When our `post` method responds with a `201` status, the `content` returned is also an `application/json` representation of the newly saved `BlogPost`.

[106] https://app.swaggerhub.com/apis/DataDaddy/spring-web_essentials_blog_api/0.2.0

Create a blog post domain class

1. From the Project pane, right-click on `com.spankinfresh.blog` under `src/main/java` then choose **New > Java Class**.
2. Enter `domain.BlogPost` as the class name and, when prompted, click "Add" to add our new class to version control.
3. Add the following properties to the class:

```
1   private long id;
2   private String category;
3   private Date datePosted;
4   private String title;
5   private String content;
```

4. Using the **Code > Generate**..., then **Getter and Setter**, add getters and setters for the properties above.
5. To make testing easier, add default and non-default constructors as shown below:

```
1   public BlogPost() { }
2
3   public BlogPost(long id, String category, Date datePosted,
4     String title, String content) {
5     this.id = id;
6     this.category = category;
7     this.datePosted = datePosted;
8     this.title = title;
9     this.content = content;
10  }
```

Incorporate the BlogPost class into our API

Update the test to pass an instance

One of the dependencies Spring Boot included is jackson-databind[107]. Spring uses it internally to marshall and unmarshal objects like BlogPost, represented as JSON in transit. This conversion is automatic in our controllers. We can use the library in our tests to avoid having to create hand-coded, long, hard to read JSON strings.

[107] https://fasterxml.github.io/jackson-databind/

1. Back in `BlogPostControllerTests`, add an instance variable for the jackson-databind object mapper[108]:

```
private final ObjectMapper mapper = new ObjectMapper();
```

2. Since most test methods will need an instance of `BlogPost` for testing, let us create a reusable constant:

```
private static final BlogPost testPosting =
  new BlogPost(1L, "category", null, "title", "content");
```

3. **Replace** the two existing tests with the one shown below. It combines the two previous tests plus sends a representation of the test blog post and checks that one is returned:

```
@Test
@DisplayName("T01 - POST accepts and returns blog post representation")
public void postCreatesNewBlogEntry_Test(@Autowired MockMvc mockMvc)
  throws Exception {
    MvcResult result = mockMvc.perform(post(RESOURCE_URI)
    .contentType(MediaType.APPLICATION_JSON)
    .content(mapper.writeValueAsString(testPosting)))
    .andExpect(status().isCreated())
    .andExpect(jsonPath("$.id").value(1L))
    .andExpect(jsonPath("$.title").value(testPosting.getTitle()))
    .andExpect(jsonPath("$.category").value(testPosting.getCategory()))
    .andExpect(jsonPath("$.content").value(testPosting.getContent()))
    .andReturn();
    MockHttpServletResponse mockResponse = result.getResponse();
    assertEquals("http://localhost/api/articles/1",
    mockResponse.getHeader("Location"));
}
```

4. When given the choice of which `jsonPath` class to import, choose, `org.springframework.test.web.servlet.re`
5. Finally, run test and verify it fails with the following message:

```
java.lang.AssertionError: No value at JSON path "$.id"
```

Before we get our test passing, let's compare it to our previous tests. The first difference is the new method calls chained to `post()` when we call `mockMvc.perform()`. The call to `contentType()` sets the `Content-type` header to `application/json`. The second is the call to `.content()` where we use

[108] https://fasterxml.github.io/jackson-databind/javadoc/2.7/com/fasterxml/jackson/databind/ObjectMapper.html

our jackson-databind object mapper to marshall the testPosting constant, created earlier, to JSON. It sets the request body with the return value from `mapper.writeValueAsString(testPosting)`.

If you were to print it out, the POSTed request body would look like this:

```
{
  "id": 1,
  "category": "Some category",
  "title": "Ab fab title",
  "content": "Amazing content"
}
```

The final difference is the additional calls to `.andExpect()` where we use MockMvcResultMatchers[109], specifically the JsonPathResultMatchers[110], allowing us to examine JSON values in the response body using JsonPath[111] expressions.

Our test expects the response body to be the same as the request body. In the next chapter, we will start saving postings to the database, and it will assign the ID. At that time, we will update our test to omit the ID in the request and to verify a non-zero ID value in the response.

Receive and return the blog posting

The changes to the controller method are pretty straightforward. We need to add a `blogPost` parameter for the @RequestBody[112]. Next, we need to replace the empty string we were returning with the parameter passed. Finally, we should practice good type safety hygiene by specifying the BlogPost type for the parameterized ResponseEntity class.

Below is the updated code:

```
  @PostMapping
  public ResponseEntity<BlogPost> createBlogEntry(
    @RequestBody BlogPost blogPost) {
      HttpHeaders headers = new HttpHeaders();
      headers.add("Location", "http://localhost/api/articles/1");
      return new ResponseEntity<>(blogPost, headers,
        HttpStatus.CREATED);
  }
```

After making these code changes, rerun the unit test in `BlogPostControllerTests` and make sure it passes. If you kept the previous unit tests, they would fail because they aren't sending a blog post

[109] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/result/MockMvcResultMatchers.html

[110] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/result/JsonPathResultMatchers.html

[111] https://github.com/json-path/JsonPath

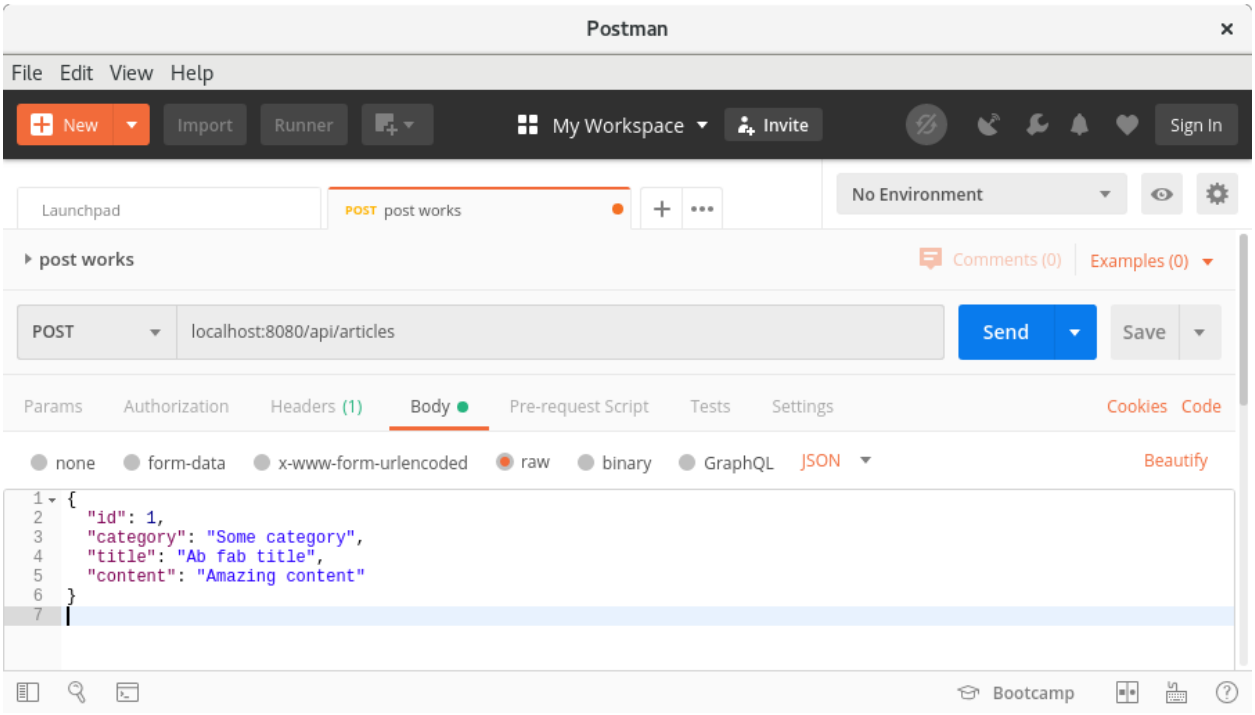
[112] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestBody.html

in the request body. Either remove them or update them also to set the content type and content values like we did for our new test.

Checking our work with Postman

1. Right-click on `BlogApplication` then choose **Run 'BlogApplication'**
2. Once the application is running, start Postman choosing the **post works** request in the **Food Blog** collection we created earlier.
3. Click on the "Body" tab under the URL address field and click on the "raw" radio button.
4. Copy or type in the JSON shown in the previous section into the request body field.
5. Change the content type dropdown to the right of GraphQL from `Text` to `JSON`. The request should look like this:

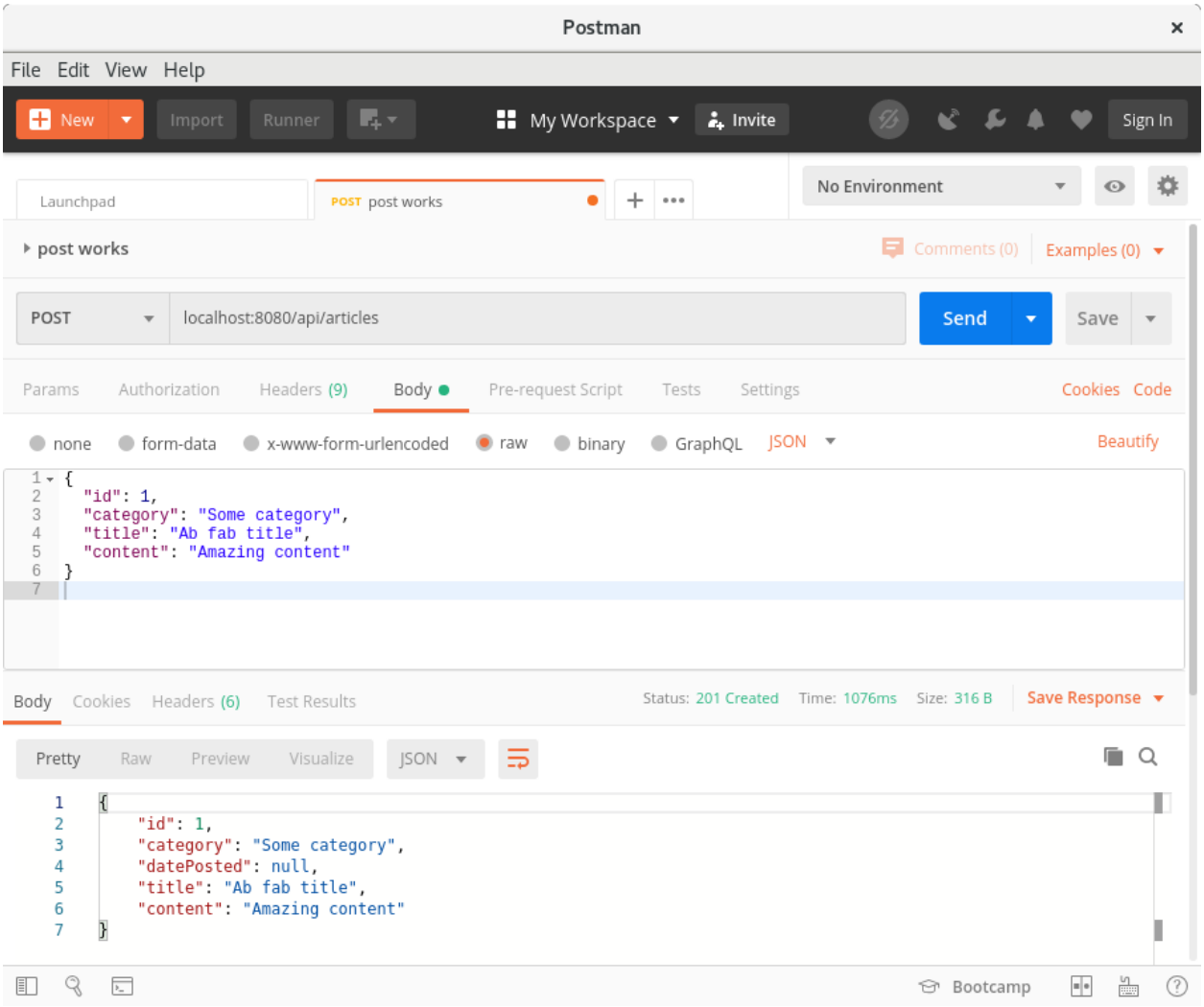
   Updated POST before sending



Updated POST before sending

6. Click "Send". After a few moments, you should get a 201 created status with the JSON blog post representation shown in the response body:

   Updated POST after sending



Updated POST after sending

Accomplishments

In this chapter, we created a domain class to represent a posting on our food blog. We updated the POST method to receive and return it. In the next chapter, we will start saving posts to the database and replace the hardcoded location header ID value with the one assigned by the database.

Adding a database

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Add the project dependencies:

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Convert BlogPost into an @Entity:

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Create a Spring JPA repository:

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Using our new JPA repository

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Update our test to check for auto-incrementing primary keys

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Saving blog posts to the database

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Have the controller automatically set the datePosted

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Write the test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Set the date in the controller:

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Test your changes with Postman

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Accomplishments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Testing with mocks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Create a second BlogPostControllerTests class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Configure our new class for mocking

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Using our mock bean

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Accomplishments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Property validation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Add bean validation constraints to BlogPost

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Test that validation errors result in a bad request status

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Update the controller to run validation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Testing with Postman

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Customizing the validation errors response

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Write a test for our custom validation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Write a custom validation exception handler

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Test your changes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Accomplishments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Putting the RUD in CRUD

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Test utilities

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Get all articles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Write the tests to get all articles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Add a controller method to get all blog articles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Checking to make sure a populated list also works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Testing with PostMan

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Get an article by ID

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Testing the 200 and 404 responses

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Add the controller method

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Update an existing article

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Tests for 204 and 404

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Getting our test to pass

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Testing our 400 errors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Testing the 409 status

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Delete an existing article

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Create our tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Implement the controller delete method

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.

Accomplishments:

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/springwebessentials.