

5

Getting Groovy with the Spring Boot CLI

This chapter covers

- Automatic dependencies and imports
- Grabbing dependencies
- Testing CLI-based applications

Some things go really well together. Peanut butter and jelly. Abbott and Costello. Thunder and lightning. Milk and cookies. On their own, these things are great. But when paired up, they're even more awesome.

So far, we've seen a lot of great things that Spring Boot has to offer, including auto-configuration and starter dependencies. When paired up with the elegance of the Groovy language, the result can be greater than the sum of its parts.

In this chapter, we're going to look at the Spring Boot CLI, a command-line tool that brings the power of Spring Boot and Groovy together to form a simple and compelling development tool for creating Spring applications. To demonstrate the power of Spring Boot's CLI, we're going to rewind the reading-list application from chapter 2, rewriting it from scratch in Groovy and taking advantage of the benefits that the CLI has to offer.

5.1 Developing a Spring Boot CLI application

Most development projects that target the JVM platform are developed in the Java language and involve a build system such as Maven or Gradle to produce a deployable artifact. In fact, the reading-list application we created in chapter 2 follows this model.

The Java language has seen great improvements in recent versions. Even so, Java has a few strict rules that add noise to the code. Line-ending semicolons, class and method modifiers (such as `public` and `private`), getter and setter methods, and `import` statements serve a purpose in Java, but they distract from the essentials of the code. From a developer's perspective, code noise is friction—friction when writing code and even more friction when trying to read it. If some of this code noise could be eliminated, it'd be easier to develop and read the code.

Likewise, build systems such as Maven and Gradle serve a purpose in a project. But build specifications are also one more thing that must be developed and maintained. If only there were a way to do away with the build, projects would be that much simpler.

When working with the Spring Boot CLI, there is no build specification. The code itself serves as the build specification, providing hints that guide the CLI in resolving dependencies and producing deployment artifacts. Moreover, by teaming up with Groovy, the Spring Boot CLI offers a development model that eliminates almost all code noise, producing a friction-free development experience.

In the very simplest case, writing a CLI-based application could be as easy as writing a single standalone Groovy script like the one we wrote in chapter 1. But when writing a more complete application with the CLI, it makes sense to set up a basic project structure to house the project code. That's where we'll get started with rewriting the reading-list application.

5.1.1 Setting up the CLI project

The first thing we'll do is create a directory structure to house the project. Unlike Maven- and Gradle-based projects, Spring Boot CLI projects don't have a strict project structure. In fact, the simplest Spring Boot CLI app could be a single Groovy script living in any directory in the filesystem. For the reading-list project, however, you should create a fresh, clean directory to keep the code separate from anything else you keep on your machine:

```
$ mkdir readinglist
```

Here I've named the directory "readinglist", but feel free to name it however you wish. The name isn't as important as the fact that the project will have a place to live.

We'll need a couple of extra directories to hold the static web content and the Thymeleaf template. Within the readinglist directory, create two new directories named "static" and "templates":

```
$ cd readinglist
$ mkdir static
$ mkdir templates
```

It's no coincidence that these directories are named the same as the directories we created under `src/main/resources` in the Java-based project. Although Spring Boot doesn't enforce a project structure like Maven and Gradle do, Spring Boot will still auto-configure a Spring `ResourceHttpRequestHandler` that looks for static content in a directory named "static" (among other locations). It will also still configure Thymeleaf to resolve templates from a directory named "templates".

Speaking of static content and Thymeleaf templates, those files will be exactly the same as the ones we created in chapter 2. So that you don't have to worry about remembering them later, go ahead and copy `style.css` into the static directory and `readingList.html` into templates.

At this point the reading-list project should be structured like this:

```
.
├── static
│   └── style.css
└── templates
    └── readingList.html
```

Now that the project is set up, we're ready to start writing some Groovy code.

5.1.2 *Eliminating code noise with Groovy*

On its own, Groovy is a very elegant language. Unlike Java, Groovy doesn't require qualifiers such as `public` and `private`. Nor does it demand semicolons at the end of each line. Moreover, thanks to Groovy's simplified property syntax ("GroovyBeans"), the `JavaBean` standard accessor methods are unnecessary.

Consequently, writing the `Book` domain class in Groovy is extremely easy. Create a new file at the root of the reading-list project named `Book.groovy` and write the following Groovy class in it.

```
class Book {
    Long id
    String reader
    String isbn
    String title
    String author
    String description
}
```

As you can see, this Groovy class is a mere fraction of the size of its Java counterpart. There are no setter or getter methods, no `public` or `private` modifiers, and no semicolons. The code noise that is so common in Java is squelched, and all that's left is what describes the essence of a book.

JDBC vs. JPA in the Spring Boot CLI

One difference you may have noticed between this Groovy implementation of `Book` and the Java implementation in chapter 2 is that there are no JPA annotations. That's because this time we're going to use Spring's `JdbcTemplate` to access the database instead of Spring Data JPA.

There are a couple of very good reasons why I chose JDBC instead of JPA for this example. First, by mixing things up a little, I can show off a few more auto-configuration tricks that Spring Boot performs when working with Spring's `JdbcTemplate`. But perhaps the most important reason I chose JDBC is that Spring Data JPA requires a `.class` file when generating on-the-fly implementations of repository interfaces. When you run Groovy scripts via the command line, the CLI compiles the scripts in memory and doesn't produce `.class` files. Therefore, Spring Data JPA doesn't work well when running scripts through the CLI.

That said, the CLI isn't completely incompatible with Spring Data JPA. If you use the CLI's `jar` command to package your application into a JAR file, the resulting JAR file will contain compiled `.class` files for all of your Groovy scripts. Building and running a JAR file from the CLI is a handy option when you want to deploy an application developed with the CLI, but it isn't as convenient during development when you want to see the results of your work quickly.

Now that we've defined the `Book` domain class, let's write the repository. First, let's write the `ReadingListRepository` interface (in `ReadingListRepository.groovy`):

```
interface ReadingListRepository {  
  
    List<Book> findByReader(String reader)  
  
    void save(Book book)  
  
}
```

Aside from a clear lack of semicolons and no `public` modifier on the interface, the Groovy version of `ReadingListRepository` isn't much different from its Java counterpart. The most significant difference is that it doesn't extend `JpaRepository` because we're not working with Spring Data JPA in this chapter. And since we're not using Spring Data JPA, we're going to have to write the implementation of `ReadingListRepository` ourselves. The following listing shows what `JdbcReadingListRepository.groovy` should look like.

Listing 5.1 A Groovy and JDBC implementation of `ReadingListRepository`

```
@Repository  
class JdbcReadingListRepository implements ReadingListRepository {  
  
    @Autowired
```

```

JdbcTemplate jdbc
List<Book> findByReader(String reader) {
    jdbc.query(
        "select id, reader, isbn, title, author, description " +
        "from Book where reader=?",
        { rs, row ->
            new Book(id: rs.getLong(1),
                    reader: rs.getString(2),
                    isbn: rs.getString(3),
                    title: rs.getString(4),
                    author: rs.getString(5),
                    description: rs.getString(6))
        } as RowMapper,
        reader)
}

void save(Book book) {
    jdbc.update("insert into Book " +
        "(reader, isbn, title, author, description) " +
        "values (?, ?, ?, ?, ?)",
        book.reader,
        book.isbn,
        book.title,
        book.author,
        book.description)
}
}

```

For the most part, this is a typical `JdbcTemplate`-based repository implementation. It's injected, via autowiring, with a reference to a `JdbcTemplate` object that it uses to query the database for books (in the `findByReader()` method) and to save books to the database (in the `save()` method).

By writing it in Groovy, we're able to apply some Groovy idioms in the implementation. For example, in `findByReader()`, a Groovy closure is given as a parameter in the call to `query()` in place of a `RowMapper` implementation.¹ Also, within the closure, a new `Book` object is created using Groovy's support for setting object properties at construction.

While we're thinking about database persistence, we also need to create a file named `schema.sql` that will contain the SQL necessary to create the `Book` table that the repository issues queries against:

```

create table Book (
    id identity,
    reader varchar(20) not null,
    isbn varchar(10) not null,
    title varchar(50) not null,
    author varchar(50) not null,
    description varchar(2000) not null
);

```

¹ In fairness to Java, we can do something similar in Java 8 using lambdas (and method references).

I'll explain how `schema.sql` is used later. For now, just know that you need to create it at the root of the classpath (at the root directory of the project) so that there will actually be a `Book` table to query against.

All of the Groovy pieces are coming together, but there's one more Groovy class we must write to make the Groovy-ified reading-list application complete. We need to write a Groovy implementation of `ReadingListController` to handle web requests and serve the reading list to the browser. At the root of the project, create a file named `ReadingListController.groovy` with the following content.

Listing 5.2 `ReadingListController` handles web requests for displaying and adding

```
@Controller
@RequestMapping("/")
class ReadingListController {

    String reader = "Craig"

    @Autowired
    ReadingListRepository readingListRepository

    @RequestMapping(method=RequestMethod.GET)
    def readersBooks(Model model) {
        List<Book> readingList =
            readingListRepository.findByReader(reader)

        if (readingList) {
            model.addAttribute("books", readingList)

            "readingList"
        }

        @RequestMapping(method=RequestMethod.POST)
        def addToReadingList(Book book) {
            book.setReader(reader)
            readingListRepository.save(book)
            "redirect:/"
        }
    }
}
```

Annotations and actions in the code:

- Inject ReadingListRepository**: Points to the `@Autowired` annotation.
- Fetch reading list**: Points to the `readingListRepository.findByReader(reader)` call.
- Populate model**: Points to the `model.addAttribute("books", readingList)` call.
- Return view name**: Points to the `"readingList"` string.
- Save book**: Points to the `readingListRepository.save(book)` call.
- Redirect after POST**: Points to the `"redirect:/"` string.

Comparing this version of `ReadingListController` with the one from chapter 2, it's easy to see that there's a lot in common. The main difference, once again, is that Groovy's syntax does away with class and method modifiers, semicolons, accessor methods, and other unnecessary code noise.

You'll also notice that both handler methods are declared with `def` rather than `String` and both dispense with an explicit `return` statement. If you prefer explicit typing on the methods and explicit `return` statements, feel free to include them—Groovy won't mind.

There's one more thing we need to do before we can run the application. Create a new file named `Grabs.groovy` and put these three lines in it:

```
@Grab("h2")
@Grab("spring-boot-starter-thymeleaf")
class Grabs {}
```

We'll talk more about what this class does later. For now, just know that the `@Grab` annotations on this class tell Groovy to fetch a few dependency libraries on the fly as the application is started.

Believe it or not, we're ready to run the application. We've created a project directory, copied a stylesheet and Thymeleaf template into it, and filled it with Groovy code. All that's left is to run it with the Spring Boot CLI (from within the project directory):

```
$ spring run .
```

After a few seconds, the application should be fully started. Open your web browser and navigate to `http://localhost:8080`. Assuming everything goes well, you should see the same reading-list application you saw in chapter 2.

Success! In just a few pages of this book, you've written a complete (albeit simple) Spring application!

At this point, however, you might be wondering how it works, considering that...

- *There's no Spring configuration.* How are the beans created and wired together? Where does the `JdbcTemplate` bean come from?
- *There's no build file.* Where do the library dependencies like Spring MVC and Thymeleaf come from?
- *There are no import statements.* How can Groovy resolve types like `JdbcTemplate` and `RequestMapping` if there are no import statements to specify what packages they're in?
- *We never deployed the app.* Where'd the web server come from?

Indeed, the code we've written seems to be missing more than just a few semicolons. How does this code even work?

5.1.3 *What just happened?*

As you've probably surmised, there's more to Spring Boot's CLI than just a convenient means of writing Spring applications with Groovy. The Spring Boot CLI has several tricks in its repertoire, including the following:

- The CLI is able to leverage Spring Boot auto-configuration and starter dependencies.
- The CLI is able to detect when certain types are in use and automatically resolve the appropriate dependency libraries to support those types.
- The CLI knows which packages several commonly used types are in and, if those types are used, adds those packages to Groovy's default packages.

- By applying both automatic dependency resolution and auto-configuration, the CLI can detect that it's running a web application and automatically include an embedded web container (Tomcat by default) to serve the application.

If you think about it, these are the most important features that the CLI offers. The Groovy syntax is just a bonus!

When you run the reading-list application through the Spring Boot CLI, several things happen under the covers to make this magic work. One of the very first things the CLI does is attempt to compile the Groovy code using an embedded Groovy compiler. Without you knowing it, however, the code fails to compile due to several unknown types in the code (such as `JdbcTemplate`, `Controller`, `RequestMapping`, and so on).

But the CLI doesn't give up. The CLI knows that `JdbcTemplate` can be added to the classpath by adding the Spring Boot JDBC starter as a dependency. It also knows that the Spring MVC types can be found by adding the Spring Boot web starter as a dependency. So it grabs those dependencies from the Maven repository (Maven Central, by default).

If the CLI were to try to recompile at this point, it would still fail because of the missing import statements. But the CLI also knows the packages of many commonly used types. Taking advantage of the ability to customize the Groovy compiler's default package imports, the CLI adds all of the necessary packages to the Groovy compiler's default imports list.

Now it's time for the CLI to attempt another compile. Assuming there are no other problems outside of the CLI's abilities (such as syntax errors or types that the CLI doesn't know about), the code will compile cleanly and the CLI will run it via an internal bootstrap method similar to the `main()` method we put in `Application` for the Java-based example.

At this point, Spring Boot auto-configuration kicks in. It sees that Spring MVC is on the classpath (as a result of the CLI resolving the web starter), so it automatically configures the appropriate beans to support Spring MVC, as well as an embedded Tomcat bean to serve the application. It also sees that `JdbcTemplate` is on the classpath, so it automatically creates a `JdbcTemplate` bean, wiring it with a `DataSource` bean that was also automatically created.

Speaking of the `DataSource` bean, it's just one of several other beans that are created via Spring Boot auto-configuration. Spring Boot also automatically configures beans that support Thymeleaf views in Spring MVC. This happens because we used `@Grab` to add H2 and Thymeleaf to the classpath, which triggers auto-configuration for an embedded H2 database and Thymeleaf.

The `@Grab` annotation is an easy way to add dependencies that the CLI isn't able to automatically resolve. In spite of its ease of use, however, there's more to this little annotation than meets the eye. Let's take a closer look at `@Grab` to see what makes it tick, how the Spring Boot CLI makes it even easier by requiring only an artifact name for many commonly used dependencies, and how to configure its dependency-resolution process.

5.2 Grabbing dependencies

In the case of Spring MVC and `JdbcTemplate`, Groovy compilation errors triggered the Spring Boot CLI to go fetch the necessary dependencies and add them to the class-path. But what if a dependency is required but there's no failing code to trigger automatic dependency resolution? Or what if the required dependency isn't among the ones the CLI knows about?

In the reading-list example, we needed Thymeleaf libraries so that we could write our views using Thymeleaf templates. And we needed the H2 database library so that we could have an embedded H2 database. But because none of the Groovy code directly referenced Thymeleaf or H2 classes, there were no compilation failures to trigger them to be resolved automatically. Therefore, we had to help the CLI out a bit by adding the `@Grab` dependencies in the `Grabs` class.

WHERE SHOULD YOU PLACE @GRAB? It's not strictly necessary to put the `@Grab` annotations in a separate class as we have. They would still do their magic had we put them in `ReadingListController` or `JdbcReadingListRepository`. For organization's sake, however, it's useful to create an otherwise empty class definition that has all of the `@Grab` annotations. This makes it easy to view all of the explicitly declared library dependencies in one place.

The `@Grab` annotation comes from Groovy's Grape (Groovy Adaptable Packaging Engine or Groovy Advanced Packaging Engine) facility. In a nutshell, Grape enables Groovy scripts to download dependency libraries at runtime without using a build tool like Maven or Gradle. In addition to providing the functionality behind the `@Grab` annotation, Grape is also used by the Spring Boot CLI to fetch dependencies deduced from the code.

Using `@Grab` is as simple as expressing the dependency coordinates. For example, suppose you want to add the H2 database to your project. Adding the following `@Grab` to one of the project's Groovy scripts will do just that:

```
@Grab(group="com.h2database", module="h2", version="1.4.190")
```

Used this way, the `group`, `module`, and `version` attributes explicitly specify the dependency. Alternatively, you can express the same dependency more succinctly using a colon-separated form similar to how dependencies can be expressed in a Gradle build specification:

```
@Grab("com.h2database:h2:1.4.185")
```

These are two textbook examples of using `@Grab`. But the Spring Boot CLI extends `@Grab` in a couple of ways to make working with `@Grab` even easier.

First, for many dependencies it's unnecessary to specify the version. Applying this to the example of the H2 database dependency, it's possible to express the dependency with the following `@Grab`:

```
@Grab("com.h2database:h2")
```

The specific version of the dependency is determined by the version of the CLI that you're using. In the case of Spring Boot CLI 1.3.0.RELEASE, the H2 dependency resolved will be 1.4.190.

But that's not all. For many commonly used dependencies, it's also possible to leave out the group ID, expressing the dependency by only giving the module ID to `@Grab`. This is what enabled us to express the following `@Grab` for H2 in the previous section:

```
@Grab("h2")
```

How can you know which dependencies require a group ID and version and which you can grab using only the module ID? I've included a complete list of all the dependencies the Spring Boot CLI knows about in appendix D. But generally speaking, it's easy enough to try `@Grab` dependencies with only a module ID first and then only express the group ID and version if the module ID alone doesn't work.

Although it's very convenient to express dependencies giving only their module IDs, what if you disagree with the version chosen by Spring Boot? What if one of Spring Boot's starters transitively pulls in a certain version of a library, but you'd prefer to use a newer version that contains a bug fix?

5.2.1 Overriding default dependency versions

Spring Boot brings a new `@GrabMetadata` annotation that can be used with `@Grab` to override the default dependency versions in a properties file.

To use `@GrabMetadata`, add it to one of the Groovy script files giving it the coordinates for a properties file with the overriding dependency metadata:

```
@GrabMetadata("com.myorg:custom-versions:1.0.0")
```

This will load a properties file named `custom-versions.properties` from a Maven repository in the `com/myorg` directory. Each line in the properties file should have a group ID and module ID as the key, and the version as the value. For example, to override the default version for H2 with 1.4.186, you can point `@GrabMetadata` at a properties file containing the following line:

```
com.h2database:h2=1.4.186
```

Using the Spring IO platform

One way you might want to use `@GrabMetadata` is to work with dependency versions defined in the Spring IO platform (<http://platform.spring.io/platform/>). The Spring IO platform offers a curated set of dependencies and versions aimed to give confidence in knowing which versions of Spring and other libraries will work well together. The dependencies and versions specified by the Spring IO platform is a superset of Spring Boot's set of known dependency libraries, and it includes several third-party libraries that are frequently used in Spring applications.

(continued)

If you'd like to build Spring Boot CLI applications on the Spring IO platform, you'll just need to annotate one of your Groovy scripts with the following `@GrabMetadata`:

```
@GrabMetadata('io.spring.platform:platform-versions:1.0.4.RELEASE')
```

This overrides the CLI's set of default dependency versions with those defined by the Spring IO platform.

One question you might have is where Grape fetches all of its dependencies from? And is that configurable? Let's see how you can manage the set of repositories that Grape draws dependencies from.

5.2.2 ***Adding dependency repositories***

By default, `@Grab`-declared dependencies are fetched from the Maven Central repository (<http://repo1.maven.org/maven2/>). In addition, Spring Boot also registers Spring's milestone and snapshot repositories to be able to fetch pre-released dependencies for Spring projects. For many projects, this is perfectly sufficient. But what if your project needs a library that isn't in Central or the Spring repositories? Or what if you're working within a corporate firewall and must use an internal repository?

No problem. The `@GrabResolver` annotation enables you to specify additional repositories from which dependencies can be fetched.

For example, suppose you want to use the latest Hibernate release. Recent Hibernate releases can only be found in the JBoss repository, so you'll need to add that repository via `@GrabResolver`:

```
@GrabResolver(name='jboss', root=
    'https://repository.jboss.org/nexus/content/groups/public-jboss')
```

Here the resolver is named "jboss" with the `name` attribute. The URL to the repository is specified in the `root` attribute.

You've seen how Spring Boot's CLI compiles your code and automatically resolves several known dependency libraries as needed. And with support for `@Grab` to resolve any dependencies that the CLI isn't able to resolve automatically, CLI-based applications have no need for a Maven or Gradle build specification (as is required by traditionally developed Java applications). But resolving dependencies and compiling code aren't the only things that build processes do. Project builds also usually execute automated tests. If there's no build specification, how do the tests run?

5.3 ***Running tests with the CLI***

Tests are an important part of any software project, and they aren't overlooked by the Spring Boot CLI. Because CLI-based applications don't involve a traditional build system, the CLI offers a test command for running tests.

Before you can try out the test command, you need to write a test. Tests can reside anywhere in the project, but I recommend keeping them separate from the main components of the project by putting them in a subdirectory. You can name the subdirectory anything you want, but I chose to name it “tests”:

```
$ mkdir tests
```

Within the tests directory, create a new Groovy script named `ReadingListControllerTest.groovy` and write a test for the `ReadingListController`. To get started, listing 5.3 has a single test method for testing that the controller handles HTTP GET requests properly.

Listing 5.3 A Groovy test for `ReadingListController`

```
import org.springframework.test.web.servlet.MockMvc
import static
    org.springframework.test.web.servlet.setup.MockMvcBuilders.*
import static org.springframework.test.web.servlet.request.
    MockMvcRequestBuilders.*
import static org.springframework.test.web.servlet.result.
    MockMvcResultMatchers.*
import static org.mockito.Mockito.*

class ReadingListControllerTest {

    @Test
    void shouldReturnReadingListFromRepository() {
        List<Book> expectedList = new ArrayList<Book>()
        expectedList.add(new Book(
            id: 1,
            reader: "Craig",
            isbn: "9781617292545",
            title: "Spring Boot in Action",
            author: "Craig Walls",
            description: "Spring Boot in Action is ..."
        ))

        def mockRepo = mock(ReadingListRepository.class)
        when(mockRepo.findByReader("Craig")).thenReturn(expectedList)

        def controller =
            new ReadingListController(readingListRepository: mockRepo)

        MockMvc mvc = standaloneSetup(controller).build()
        mvc.perform(get("/"))
            .andExpect(view().name("readingList"))
            .andExpect(model().attribute("books", expectedList))
    }
}
```

Mock ReadingListRepository

Perform and test GET request

As you can see, this is a simple JUnit test that uses Spring's support for mock MVC testing to fire a GET request at the controller. It starts by setting up a mock implementation of `ReadingListRepository` that will return a single-entry list of `Book`. Then it creates an instance of `ReadingListController`, injecting the mock repository into the `readingListRepository` property. Finally, it sets up a `MockMvc` object, performs a GET request, and asserts expectations with regard to the view name and model contents.

But the specifics of the test aren't as important here as how you run the test. Using the CLI's test command, you can execute the test from the command line like this:

```
$ spring test tests/ReadingListControllerTest.groovy
```

In this case, I'm explicitly selecting `ReadingListControllerTest` as the test to run. If you have several tests within the `tests/` directory and want to run them all, you can give the directory name to the test command:

```
$ spring test tests
```

If you're inclined to write Spock specifications instead of JUnit tests, you may be pleased to know that the CLI's test command can also execute Spock specifications, as demonstrated by `ReadingListControllerSpec` in the following listing.

Listing 5.4 A Spock specification to test `ReadingListController`

```
import org.springframework.test.web.servlet.MockMvc
import static
    org.springframework.test.web.servlet.setup.MockMvcBuilders.*
import static org.springframework.test.web.servlet.request.
    MockMvcRequestBuilders.*
import static org.springframework.test.web.servlet.result.
    MockMvcResultMatchers.*
import static org.mockito.Mockito.*

class ReadingListControllerSpec extends Specification {

    MockMvc mockMvc
    List<Book> expectedList

    def setup() {
        expectedList = new ArrayList<Book>()
        expectedList.add(new Book(
            id: 1,
            reader: "Craig",
            isbn: "9781617292545",
            title: "Spring Boot in Action",
            author: "Craig Walls",
            description: "Spring Boot in Action is ..."
        ))

        def mockRepo = mock(ReadingListRepository.class)
        when(mockRepo.findByReader("Craig")).thenReturn(expectedList)
    }
}
```

**Mock
ReadingListRepository**

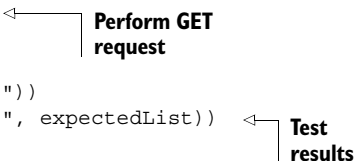


```

def controller =
    new ReadingListController(readingListRepository: mockRepo)
mockMvc = standaloneSetup(controller).build()
}

def "Should put list returned from repository into model"() {
    when:
        def response = mockMvc.perform(get("/"))
    then:
        response.andExpect(view().name("readingList"))
                  .andExpect(model().attribute("books", expectedList))
}
}

```



`ReadingListControllerSpec` is a simple translation of `ReadingListControllerTest` from a JUnit test into a Spock specification. As you can see, its one test very plainly states that when a GET request is performed against “/”, then the response should have a view named `readingList` and the expected list of books should be in the model at the key `books`.

Even though it’s a Spock specification, `ReadingListControllerSpec` can be run with `spring test` tests the same way as a JUnit-based test.

Once the code is written and the tests are all passing, you might want to deploy your project. Let’s see how the Spring Boot CLI can help produce a deployable artifact.

5.4 Creating a deployable artifact

In conventional Java projects based on Maven or Gradle, the build system is responsible for producing a deployment unit; typically a JAR file or a WAR file. With Spring Boot CLI, however, we’ve simply been running our application from the command line with the `spring` command.

Does that mean that if you want to deploy a Spring Boot CLI application you must install the CLI on your server and fire up the application manually from the command line? That seems awfully inconvenient (not to mention risky) when deploying to production environments.

We’ll talk more about options for deploying Spring Boot applications in chapter 8. For now, though, let me show you one more trick that the CLI has up its sleeve. From within the CLI-based `reading-list` application, issue the following at the command line:

```
$ spring jar ReadingList.jar .
```

This will package up the entire project, including all dependencies, Groovy, and an embedded Tomcat, into a single executable JAR file. Once complete, you’ll be able to run the app at the command line (without the CLI) like this:

```
$ java -jar ReadingList.jar
```

In addition to being run at the command line, the executable JAR can be deployed to several Platform-as-a-Service (PaaS) cloud platforms including Pivotal Cloud Foundry and Heroku. You'll see how in chapter 8.

5.5 **Summary**

The Spring Boot CLI takes the simplicity offered by Spring Boot auto-configuration and starter dependencies and turns it up a notch. Using the elegance of the Groovy language, the CLI makes it possible to develop Spring applications with minimal code noise.

In this chapter we completely rewrote the reading-list application from chapter 2. But this time we developed it in Groovy as a Spring Boot CLI application. You saw how the CLI makes Groovy even more elegant by automatically adding `import` statements for many commonly used packages and types. And the CLI is also able to automatically resolve several dependency libraries.

For libraries that the CLI is unable to automatically resolve, CLI-based applications can take advantage of the Grape `@Grab` annotation to explicitly declare dependencies without a build specification. Spring Boot's CLI extends `@Grab` so that, for many commonly needed library dependencies, you only need to declare the module ID.

Finally, you also saw how to execute tests and build deployable artifacts, tasks commonly handled by build systems, with the Spring Boot CLI.

Spring Boot and Groovy go well together, each boosting the other's simplicity. We're going to take another look at how Spring Boot and Groovy play well together in the next chapter as we explore how Spring Boot is at the core of the latest version of Grails.

Applying Grails in Spring Boot

This chapter covers

- Persisting data with GORM
- Defining GSP views
- An introduction to Grails 3 and Spring Boot

When I was growing up, there was a series of television advertisements involving two people, one enjoying a chocolate bar and another eating peanut butter out of a jar. By way of some sort of comedic mishap, the two would collide, resulting in the peanut butter and chocolate getting mixed.

One would proclaim, “You got your chocolate in my peanut butter!” The other would respond, “You got peanut butter on my chocolate!”

After initially being angry with their circumstances, the two would conclude that the combination of peanut butter and chocolate is a good thing. Then a voice-over would suggest that the viewer should eat a Reese’s Peanut Butter Cup.

From the moment that Spring Boot was announced, I’ve been frequently asked how to choose between Spring Boot and Grails. Both are built upon the Spring Framework and both help ease application development. Indeed, they’re very

much like peanut butter and chocolate. Both are great, but the choice is largely a personal one.

As it turns out, there's no reason to choose one or the other. Just like the chocolate vs. peanut butter debate, Spring Boot and Grails are two great choices that work great together.

In this chapter, we're going to look at the connection between Grails and Spring Boot. We'll start by looking at a few Grails features like GORM and Groovy Server Pages (GSP) that are available in Spring Boot. Then we'll flip it around and see how Grails 3 has been reinvented by being built upon Spring Boot.

6.1 *Using GORM for data persistence*

Probably one of the most intriguing pieces of Grails is GORM (Grails object-relational mapping). GORM makes database work as simple as declaring the entities that will be persisted. For example, listing 6.1 shows how the Book entity from the reading-list example could be written in Groovy as a GORM entity.

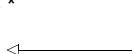
Listing 6.1 A GORM Book entity

```
package readinglist

import grails.persistence.*

@Entity
class Book {

    Reader reader
    String isbn
    String title
    String author
    String description
}
```

 This is a GORM entity

Just like its Java equivalent, this Book class has a handful of properties that describe a book. Unlike the Java version, however, it's not littered with semicolons, public or private modifiers, setter and getter methods, or any of the other noise that's common in Java. But what makes it a GORM entity is that it's annotated with the @Entity annotation from Grails. This simple entity does a lot, including mapping the object to the database and enabling Book with persistence methods through which it can be saved and retrieved.

To use GORM with a Spring Boot project, all you must do is add the GORM dependency to your build. In Maven, the <dependency> looks like this:

```
<dependency>
  <groupId>org.grails</groupId>
  <artifactId>gorm-hibernate4-spring-boot</artifactId>
  <version>1.1.0.RELEASE</version>
</dependency>
```

The same dependency can be expressed in a Gradle build like this:

```
compile("org.grails:gorm-hibernate4-spring-boot:1.1.0.RELEASE")
```

This library carries some Spring Boot auto-configuration with it that will automatically configure all of the necessary beans to support working with GORM. All you need to do is start writing the code.

Another GORM option for Spring Boot

As its name suggests, the `gorm-hibernate4-spring-boot` dependency enables GORM for data persistence via Hibernate. For many projects, this will be fine. If, however, you're interested in working with the MongoDB document database, you'll be pleased to know that GORM for MongoDB is also available for Spring Boot.

The Maven dependency looks like this:

```
<dependency>
  <groupId>org.grails</groupId>
  <artifactId>gorm-mongodb-spring-boot</artifactId>
  <version>1.1.0.RELEASE</version>
</dependency>
```

Likewise, the Gradle dependency is as follows:

```
compile("org.grails:gorm-mongodb-spring-boot:1.1.0.RELEASE")
```

Due to the nature of how GORM works, it requires that at least the entity class be written in Groovy. We've already written the `Book` entity in listing 6.1. As for the `Reader` entity, it's shown in the following listing.

Listing 6.2 A GORM Reader entity

```
package readinglist

import grails.persistence.*

import org.springframework.security.core.GrantedAuthority
import
    org.springframework.security.core.authority.SimpleGrantedAuthority
import org.springframework.security.core.userdetails.UserDetails

@Entity
class Reader implements UserDetails {
    String username
    String fullname
    String password

    Collection<? extends GrantedAuthority> getAuthorities() {
```

← This is
an entity

```

    Arrays.asList(new SimpleGrantedAuthority("READER"))
  }

  boolean isAccountNonExpired() {
    true
  }

  boolean isAccountNonLocked() {
    true
  }

  boolean isCredentialsNonExpired() {
    true
  }

  boolean isEnabled() {
    true
  }
}

```

← **Implement
UserDetails**

Now that we've written the two GORM entities for the reading-list application, we'll need to rewrite the rest of the app to use them. Because working with Groovy is such a pleasant experience (and very Grails-like), we'll continue writing the other classes in Groovy as well.

First up is `ReadingListController`, as shown next.

Listing 6.3 A Groovy reading-list controller

```

package readinglist

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.context.properties.ConfigurationProperties
import org.springframework.http.HttpStatus
import org.springframework.stereotype.Controller
import org.springframework.ui.Model
import org.springframework.web.bind.annotation.ExceptionHandler
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
import org.springframework.web.bind.annotation.ResponseStatus

@Controller
@RequestMapping("/")
@ConfigurationProperties("amazon")
class ReadingListController {

    @Autowired
    AmazonProperties amazonProperties

    @ExceptionHandler(value=RuntimeException.class)
    @ResponseStatus(value=HttpStatus.BANDWIDTH_LIMIT_EXCEEDED)
    def error() {

```

```

    "error"
}

@RequestMapping(method=RequestMethod.GET)
def readersBooks(Reader reader, Model model) {
    List<Book> readingList = Book.findAllByReader(reader)
    model.addAttribute("reader", reader)
    if (readingList) {
        model.addAttribute("books", readingList)
        model.addAttribute("amazonID", amazonProperties.getAssociateId())
    }
    "readingList"
}

@RequestMapping(method=RequestMethod.POST)
def addToReadingList(Reader reader, Book book) {
    Book.withTransaction {
        book.setReader(reader)
        book.save()
    }
    "redirect:/"
}
}

```

Find all reader books

Save a book

The most obvious difference between this version of `ReadingListController` and the one from chapter 3 is that it's written in Groovy and lacks much of the code noise from Java. But the most significant difference is that it doesn't work with an injected `ReadingListRepository` anymore. Instead, it works directly with the `Book` type for persistence.

In the `readersBooks()` method, it calls the static `findAllByReader()` method on `Book` to fetch all books for the given reader. Although we didn't write a `findAllByReader()` method in listing 6.1, this will work because GORM will implement it for us.

Likewise, the `addToReadingList()` method uses the static `withTransaction()` and the instance `save()` methods, both provided by GORM, to save a `Book` to the database.

And all we had to do was declare a few properties and annotate `Book` with `@Entity`. A pretty good payoff, if you ask me.

A similar change must be made to `SecurityConfig` to fetch a `Reader` via GORM rather than using `ReadingListRepository`. The following listing shows the new Groovy `SecurityConfig`.

Listing 6.4 SecurityConfig in Groovy

```

package readinglist

import org.springframework.context.annotation.Configuration
import org.springframework.security.config.annotation.authentication.
    builders.AuthenticationManagerBuilder
import org.springframework.security.config.annotation.web.
    builders.HttpSecurity

```

```

import org.springframework.security.config.annotation.web.
    configuration.WebSecurityConfigurerAdapter
import org.springframework.security.core.userdetails.UserDetailsService

@Configuration
class SecurityConfig extends WebSecurityConfigurerAdapter {

    void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/").access("hasRole('READER')")
                .antMatchers("/**").permitAll()
            .and()
            .formLogin()
                .loginPage("/login")
                .failureUrl("/login?error=true")
        }

    void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .userDetailsService(
                { username -> Reader.findByUsername(username) }
                as UserDetailsService)
        }
    }
}

```

← Find a reader by username

Aside from being rewritten in Groovy, the most significant change in `SecurityConfig` is the second `configure()` method. As you can see, it uses a closure (as the implementation of `UserDetailsService`) that looks up a `Reader` by calling the static `findByUsername()` method, which is provided by GORM.

You may be wondering what becomes of `ReadingListRepository` in this GORM-enabled application. With GORM handling all of the persistence for us, `ReadingListRepository` is no longer needed. Neither are any of its implementations. I think you'll agree that less code is a good thing.

As for the remaining code in the application, it should also be rewritten in Groovy to match the classes we've changed thus far. But none of it deals with GORM and is therefore out of scope for this chapter. The complete Groovy application is available in the example code download.

At this point, you can fire up the reading-list application using any of the ways we've already discussed for running Spring Boot applications. Once it starts, the application should work as it always has. Only you and I know that the persistence mechanism has been changed.

In addition to GORM, Grails apps usually use Groovy Server Pages to render model data as HTML served to the browser. The Grails-ification of our application continues in the next section, where we'll replace the Thymeleaf templates with equivalent GSP.

6.2 Defining views with Groovy Server Pages

Up until now, we've been using Thymeleaf templates to define the view for the reading-list application. In addition to Thymeleaf, Spring Boot also offers Freemarker, Velocity, and Groovy-based templates. For any of those choices, all you must do is add the appropriate starter to your build and start writing templates in the `templates/` directory at the root of the classpath. Auto-configuration takes care of the rest.

The Grails project also offers auto-configuration for Groovy Server Pages (GSP). If you want to use GSP in your Spring Boot application, all you must do is add the GSP for Spring Boot library to your build:

```
compile("org.grails:grails-gsp-spring-boot:1.0.0")
```

Just like the other view template options offered by Spring Boot, simply having this library in your classpath triggers auto-configuration that sets up the view resolvers necessary for GSP to work as the view layer of Spring MVC.

All that's left is to write the GSP templates for your application. For the reading-list application, we'll need to rewrite the Thymeleaf `readingList.html` file in GSP form as `readingList.gsp` (in `src/main/resources/templates`). The following listing shows the new GSP-enabled reading-list template.

Listing 6.5 The reading-list app's main view written in GSP

```
<!DOCTYPE html>
<html>
  <head>
    <title>Reading List</title>
    <link rel="stylesheet" href="/style.css"></link>
  </head>

  <body>
    <h2>Your Reading List</h2>

    <g:if test="${books}">
      <g:each in="${books}" var="book">
        <dl>
          <dt class="bookHeadline">
            ${book.title} by ${book.author}
            (ISBN: ${book.isbn})
          </dt>
          <dd class="bookDescription">
            <g:if test="book.description">
              ${book.description}
            </g:if>
            <g:else>
              No description available
            </g:else>
          </dd>
        </dl>
      </g:each>
```

← List the books

```

</g:if>
<g:else>
  <p>You have no books in your book list</p>
</g:else>

<hr/>

<h3>Add a book</h3>

<form method="POST">
  <label for="title">Title:</label>
  <input type="text" name="title"
        value="${book?.title}"/><br/>
  <label for="author">Author:</label>
  <input type="text" name="author"
        value="${book?.author}"/><br/>
  <label for="isbn">ISBN:</label>
  <input type="text" name="isbn"
        value="${book?.isbn}"/><br/>
  <label for="description">Description:</label><br/>
  <textarea name="description" rows="5" cols="80">
    ${book?.description}
  </textarea>
  <input type="hidden" name="${_csrf.parameterName}"
        value="${_csrf.token}"/>
  <input type="submit" value="Add Book" />
</form>

</body>
</html>

```

The book form

The CSRF token

As you can see, the GSP template is sprinkled with expression language references (the parts wrapped in `${}`) and tags from the GSP tag library such as `<g:if>` and `<g:each>`. It's not quite pure HTML as is the case with Thymeleaf, but it's a familiar and comfortable option if you're used to working with JSP.

For the most part, it's rather straightforward to map the elements on this GSP template with the corresponding Thymeleaf templates from chapters 2 and 3. One thing to note, however, is that you have to put in a hidden field to carry the CSRF (Cross-Site Request Forgery) token. Spring Security requires this token on POST requests, and Thymeleaf is able to automatically include it in the rendered HTML. With GSP, however, you must explicitly include the CSRF token in a hidden field.

Figure 6.1 shows the results of the GSP rendered as HTML in the browser after a few books have been entered.

Although Grails features like GORM and GSP are appealing and go a long way toward making a Spring Boot application even simpler, it's not quite the complete Grails experience. We've seen how to put a little Grails chocolate in the Spring Boot peanut butter. Now we'll turn it around and see how Grails 3 gives you the best of both worlds: a development experience that's both fully Spring Boot and fully Grails.

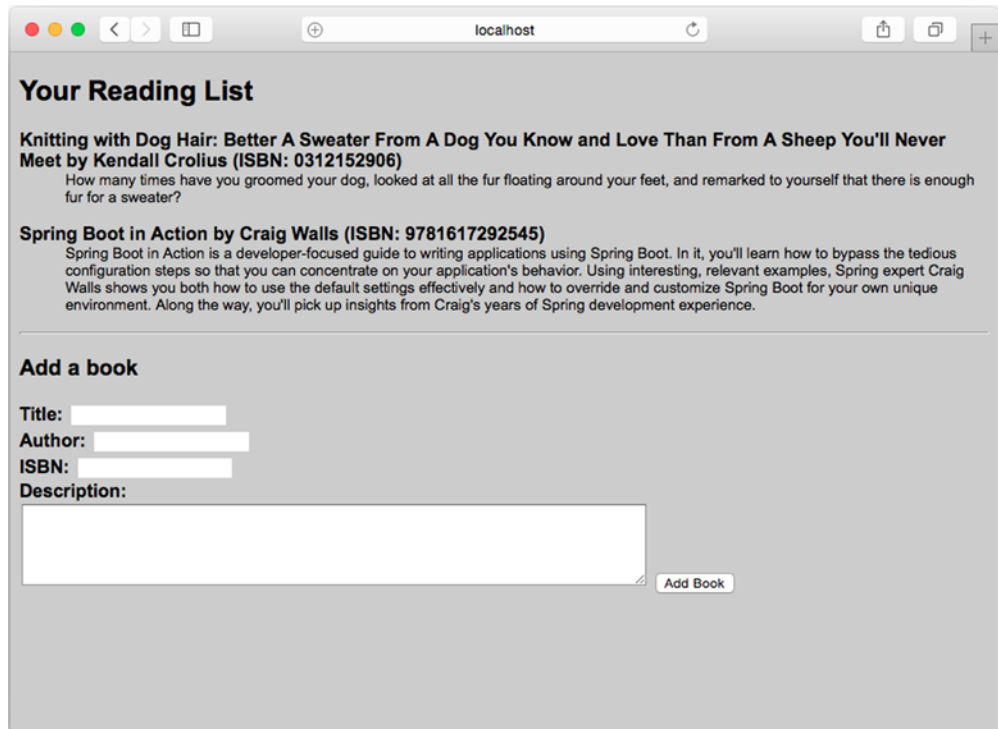


Figure 6.1 The reading list rendered from a GSP template

6.3 Mixing Spring Boot with Grails 3

Grails has always been a higher-level framework built upon the giants of Spring, Groovy, Hibernate, and others. With Grails 3, Grails is now built upon Spring Boot, enabling a very compelling developer experience that makes both Grails developers and Spring Boot developers feel at home.

The first step toward working with Grails 3 is to install it. On Mac OS X and most Unix systems, the easiest way to install Grails is to use SDKMAN at the command line:

```
$ sdk install grails
```

If you're using Windows or otherwise can't use SDKMAN, you'll need to download the binary distribution, unzip it, and add the bin directory to your system path.

Whichever installation choice you use, you can verify the installation by checking the Grails version at the command line:

```
$ grails -version
```

Assuming the installation went well, you're now ready to start creating a Grails project.

6.3.1 Creating a new Grails project

The `grails` command-line tool is what you'll use to perform many tasks with a Grails project, including the initial creation of the project. To kick off the reading-list application project, use `grails` like this:

```
$ grails create-app readinglist
```

As its name suggests, the `create-app` command creates a new application project. In this case, the name of the project is “readinglist”.

Once the `grails` tool has created the application, `cd` into the `readinglist` directory and take a look at what was created. Figure 6.2 shows a high-level view of what the project structure should look like.

You should recognize a few familiar entries in the project's directory structure. There's a Gradle build specification and configuration (`build.gradle` and `gradle.properties`). There's also a standard Gradle project structure under the `src` directory. But `grails-app` is the most interesting directory in the project. If you've ever worked with any previous version of Grails, you'll know what this directory is for. It's where you'll write the controllers, domain types, and other code that makes up the Grails project.

If you dig a little deeper and open up the `build.gradle` file, you'll find a few more familiar items. To start with, the build specification uses the Spring Boot plugin for Gradle:

```
apply plugin: "spring-boot"
```

This means that you'll be able to build and run the Grails application just as you would any other Spring Boot application.

You'll also notice that there are a handful of Spring Boot libraries among the other dependencies:

```
dependencies {
    compile 'org.springframework.boot:spring-boot-starter-logging'
    compile("org.springframework.boot:spring-boot-starter-actuator")
    compile "org.springframework.boot:spring-boot-autoconfigure"
    compile "org.springframework.boot:spring-boot-starter-tomcat"
    ...
}
```

This provides your Grails application with Spring Boot auto-configuration and logging, as well as the Actuator and an embedded Tomcat to serve the application when run as an executable JAR.

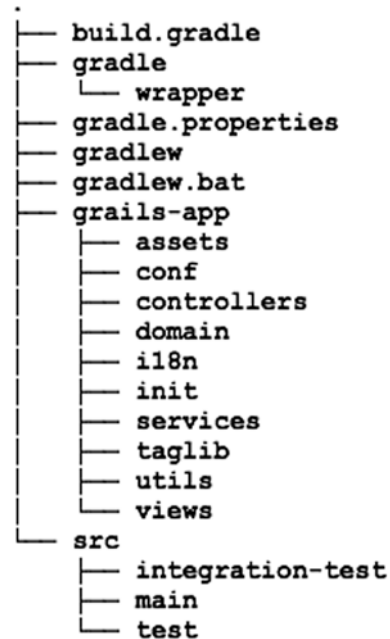


Figure 6.2 The directory structure of a Grails 3 project

Indeed, this is a Spring Boot project. It's also a Grails project. As of Grails 3, Grails is built upon a foundation of Spring Boot.

RUNNING THE APPLICATION

The most straightforward way to run a Grails application is with the `run-app` command of the `grails` tool at the command line:

```
$ grails run-app
```

Even though we've not written a single line of code, we're already able to run the application and view it in the browser. Once the application starts up, you can navigate to `http://localhost:8080` in your web browser. You should see something similar to what's shown in figure 6.3.

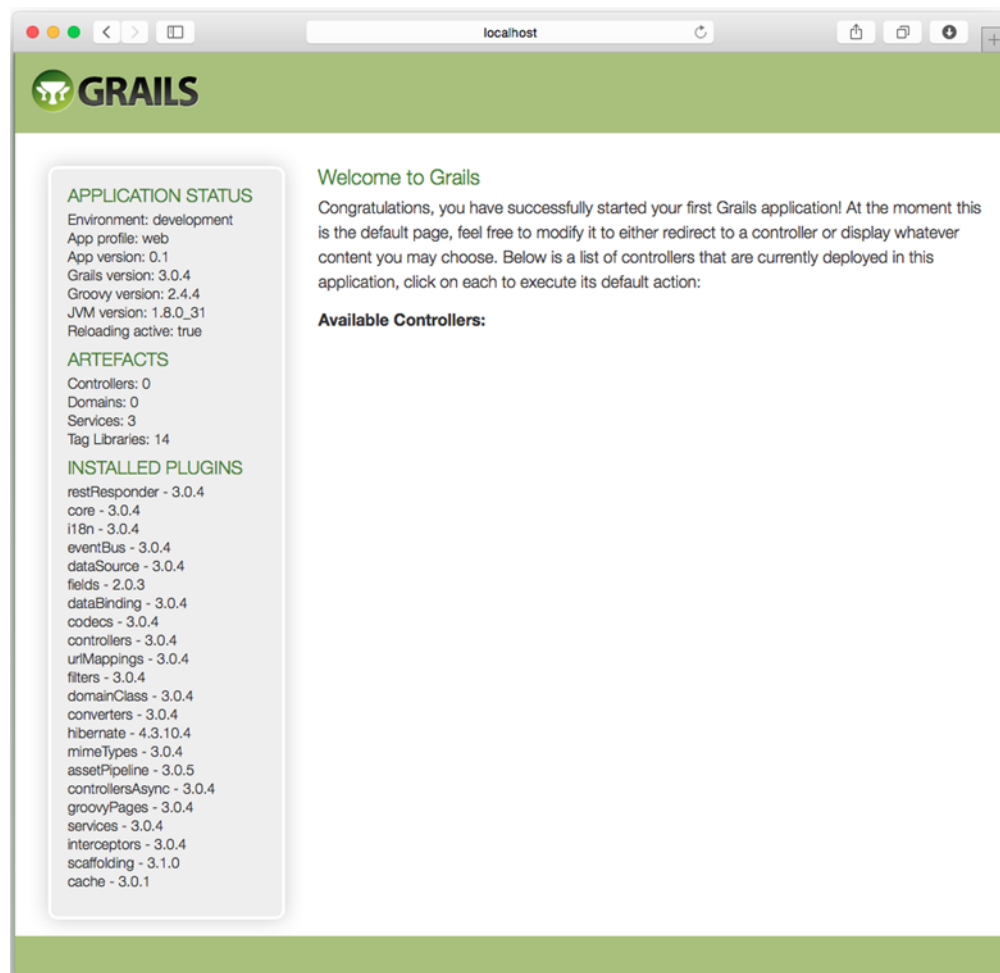


Figure 6.3 Running a freshly created Grails application

The `run-app` command is the Grails way of running the application and has been the way to run Grails applications for years, even in previous versions of Grails. But because this Grails 3 project's Gradle specification uses the Spring Boot plugin for Gradle, you can also run the application using any of the means available to a Spring Boot project. This includes the `bootRun` task via Gradle:

```
$ gradle bootRun
```

You can also build the project and run the resulting executable JAR file:

```
$ gradle build
...
$ java -jar build/lib/readingList-0.1.jar
```

Of course, the WAR file produced by the build can also be deployed to a servlet 3.0 container of your choice.

It's very convenient to be able to run the application this early in the development process. It helps you know that the project has been properly initialized. But the application doesn't do much interesting yet. It's up to us to build upon the initial project. We'll start by defining the domain.

6.3.2 *Defining the domain*

The central domain type in the reading-list application is the `Book` class. Although we could manually create a `Book.groovy` file, it's usually better to use the `grails` tool to create domain types. That's because it knows where the source files go and it's also able to generate any related artifacts for us at the same time.

To create the `Book` class, we'll use the `create-domain-class` command of the `grails` tool:

```
$ grails create-domain-class Book
```

This will generate two source files: a `Book.groovy` file and a `BookSpec.groovy` file. The latter is a Spock specification for testing the `Book` class. It's initially empty, but you can fill it with any tests you need to verify the functionality of a `Book`.

The `Book.groovy` file defines the `Book` class itself. You'll find it in `grails-app/domain/readingList`. Initially, it's rather empty and looks like this:

```
package readinglist
class Book {

    static constraints = {
    }
}
```

We'll need to add the fields that define a book, such as the title, author, and ISBN. After adding the fields, `Book.groovy` looks like this:

```
package readinglist
class Book {

    static constraints = {
    }

    String reader
    String isbn
    String title
    String author
    String description
}
```

The static constraints variable is where you can define any validation constraints to be enforced on instances of `Book`. In this chapter, we're primarily interested in building out the reading-list application to see how it's built upon Spring Boot and not so much on validation. Therefore, we'll leave the constraints empty. Feel free to add constraints if you wish, though. Have a look at *Grails in Action, Second Edition*, by Glen Smith and Peter Ledbrook (Manning, 2014) for more information.¹

For the purpose of working with Grails, we're going to keep the reading-list application simple and in line with what we wrote in chapter 2. Therefore, we'll forego creating a `Reader` domain and go ahead and create the controller.

6.3.3 Writing a Grails controller

As with domain types, it's easy to create controllers using the `grails` tool. In the case of controllers, you have a few choices of commands, however:

- `create-controller`—Creates an empty controller, leaving it to the developer to write the controller's functionality
- `generate-controller`—Generates a controller with scaffolded CRUD operations for a given domain type
- `generate-all`—Generates a scaffolded CRUD controller and associated views for a given domain type

Although scaffolded controllers are very handy and are certainly one of the most well-known features of Grails, we're going to keep it simple and write a controller that has just enough functionality to mimic the behavior of the application we created in chapter 2. Therefore, we'll use the `create-controller` command to create a bare-bones controller and then fill it in with the methods we need:

```
$ grails create-controller ReadingList
```

This command creates a controller named `ReadingListController` in `grails-app/controllers/readingList` that looks like this:

¹ Although *Grails in Action, Second Edition*, covers Grails 2, much of what you learn about Grails 2 applies to Grails 3.

```
package readinglist
class ReadingListController {

    def index() { }
}
```

Without making any changes, this controller is ready to run, although it won't do much. At this point, it will handle requests whose path is `/readingList` and forward the request to the view defined at `grails-app/views/readingList/index.gsp` (which doesn't yet exist, but we'll create soon).

But what we need our controller to do is display a list of books and a form to add a new book. We also need it to handle the form submission and save a book to the database. The following listing shows the `ReadingListController` that we need.

Listing 6.6 **Fleshing out the `ReadingListController`**

```
package readinglist

import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional

class ReadingListController {

    def index() {
        respond Book.list(params), model:[book: new Book()]
    }

    @Transactional
    def save(Book book) {
        book.reader = 'Craig'
        book.save flush:true
        redirect(action: "index")
    }
}
```

Fetch books
into model

Save the
book

Although it's much shorter than the equivalent Java controller, this version of `ReadingListController` is almost completely functionally equivalent. It handles `GET` requests for `/readingList` and fetches a list of books to be displayed. And when the form is submitted, it handles the `POST` request, saves the book, then redirects to the `index` action (which is handled by the `index()` method).

Incredibly, we're almost finished with the Grails version of the reading-list application. The only thing left is to create the view that displays the list of books and the form.

6.3.4 **Creating the view**

Grails applications typically use GSP templates for their views. You've already seen how to use GSP in a Spring Boot application, so the template we need won't be much different from the one in section 6.2.

What we might want to do, however, is take advantage of the layout facilities offered in Grails to apply a common design to all of the pages in the application. As you can see in listing 6.7, it's a rather straightforward and simple change.

Listing 6.7 A Grails-ready GSP template, including layout

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="layout" content="main"/>
    <title>Reading List</title>
    <link rel="stylesheet"
      href="/assets/main.css?compile=false" />
    <link rel="stylesheet"
      href="/assets/mobile.css?compile=false" />
    <link rel="stylesheet"
      href="/assets/application.css?compile=false" />
  </head>

  <body>
    <h2>Your Reading List</h2>

    <g:if test="${bookList && !bookList.isEmpty()}">
      <g:each in="${bookList}" var="book">
        <dl>
          <dt class="bookHeadline">
            ${book.title}</span> by ${book.author}
            (ISBN: ${book.isbn})
          </dt>
          <dd class="bookDescription">
            <g:if test="${book.description}">
              ${book.description}
            </g:if>
            <g:else>
              No description available
            </g:else>
          </dd>
        </dl>
      </g:each>
    </g:if>
    <g:else>
      <p>You have no books in your book list</p>
    </g:else>

    <hr/>

    <h3>Add a book</h3>

    <g:form action="save">
      <fieldset class="form">
        <label for="title">Title:</label>
        <g:field type="text" name="title" value="${book?.title}"/><br/>
        <label for="author">Author:</label>
        <g:field type="text" name="author">
```

Use the main layout

List the books

The book form

```

        value="${book?.author}"/><br/>
<label for="isbn">ISBN:</label>
<g:field type="text" name="isbn" value="${book?.isbn}"/><br/>
<label for="description">Description:</label><br/>
<g:textArea name="description" value="${book?.description}"
            rows="5" cols="80"/>
</fieldset>
<fieldset class="buttons">
    <g:submitButton name="create" class="save"
        value="${message(code: 'default.button.create.label',
                        default: 'Create')}" />
</fieldset>
</g:form>

</body>
</html>

```

Within the `<head>` element we've removed the `<link>` tag that references our stylesheet. In its place, we've put in a `<meta>` tag that references the "main" layout of the Grails application. As a consequence, the application will take on the Grails look and feel, as shown in figure 6.4, when you run it.

Although the Grails style is more eye-catching than the simple stylesheet we've been using, there is obviously still a little work to do to make the reading-list application look

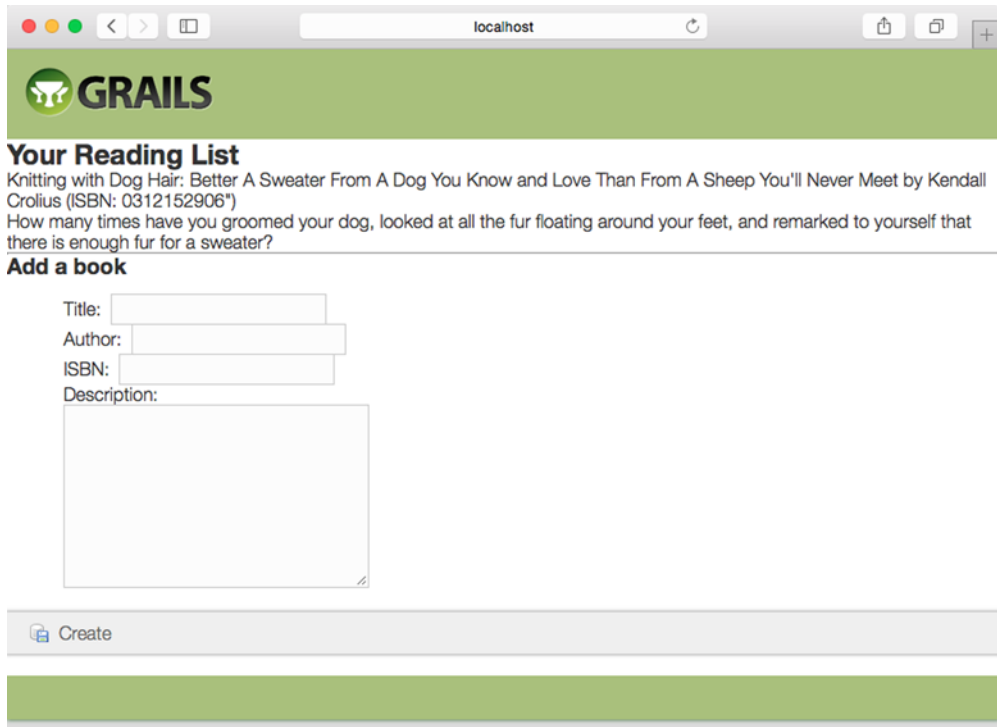


Figure 6.4 The reading-list application with the common Grails styling

good. And we'll probably want to start making the application look a little less like Grails and more like what we want our application to look like. Manipulating the application's stylesheets is well outside of the scope of this book, but if you're interested in tweaking the look and feel, you'll find the stylesheets in the `grails-app/assets/stylesheets` directory.

6.4 Summary

Both Grails and Spring Boot aim to make developers' lives easy, providing a greatly simplified development model on top of Spring, so it may appear that these are competing frameworks. But in this chapter, we've seen how to get the best of both worlds by bringing Spring Boot and Grails together.

We looked at how to add GORM and GSP views, two well-known Grails features, to an otherwise typical Spring Boot application. GORM is an especially welcome feature in Spring Boot, enabling you to perform persistence directly with the domain and eliminating the need for a repository.

Then we looked at Grails 3, the latest incarnation of Grails, built upon Spring Boot. When developing a Grails 3 application, you're also working with Spring Boot and are afforded all of the features of Spring Boot, including auto-configuration.

In all cases, both in this and the previous chapter, you've seen how mixing Groovy and Spring Boot helps squelch the code noise that's required in the Java language.

Coming up in the next chapter, we're going to shift our attention away from coding Spring Boot applications and look at the Spring Boot Actuator to see how it gives us insights into the inner workings of our running applications.



Taking a peek inside with the Actuator

This chapter covers

- Actuator web endpoints
- Adjusting the Actuator
- Shelling into a running application
- Securing the Actuator

Have you ever tried to guess what's inside a wrapped gift? You shake it, weigh it, and measure it. And you might even have a solid idea as to what's inside. But until you open it up, there's no way of knowing for sure.

A running application is kind of like a wrapped gift. You can poke at it and make reasonable guesses as to what's going on under the covers. But how can you know for sure? If only there were some way that you could peek inside a running application, see how it's behaving, check on its health, and maybe even trigger operations that influence how it runs?

In this chapter, we're going to explore Spring Boot's Actuator. The Actuator offers production-ready features such as monitoring and metrics to Spring Boot applications. The Actuator's features are provided by way of several REST endpoints,

a remote shell, and Java Management Extensions (JMX). We'll start by looking at the Actuator's REST endpoints, which offer the most complete and well-known way of working with the Actuator.

7.1 Exploring the Actuator's endpoints

The key feature of Spring Boot's Actuator is that it provides several web endpoints in your application through which you can view the internals of your running application. Through the Actuator, you can find out how beans are wired together in the Spring application context, determine what environment properties are available to your application, get a snapshot of runtime metrics, and more.

The Actuator offers a baker's dozen of endpoints, as described in table 7.1.

Table 7.1 Actuator endpoints

HTTP method	Path	Description
GET	/autoconfig	Provides an auto-configuration report describing what auto-configuration conditions passed and failed.
GET	/configprops	Describes how beans have been injected with configuration properties (including default values).
GET	/beans	Describes all beans in the application context and their relationship to each other.
GET	/dump	Retrieves a snapshot dump of thread activity.
GET	/env	Retrieves all environment properties.
GET	/env/{name}	Retrieves a specific environment value by name.
GET	/health	Reports health metrics for the application, as provided by <code>HealthIndicator</code> implementations.
GET	/info	Retrieves custom information about the application, as provided by any properties prefixed with <code>info</code> .
GET	/mappings	Describes all URI paths and how they're mapped to controllers (including Actuator endpoints).
GET	/metrics	Reports various application metrics such as memory usage and HTTP request counters.
GET	/metrics/{name}	Reports an individual application metric by name.
POST	/shutdown	Shuts down the application; requires that <code>endpoints.shutdown.enabled</code> be set to <code>true</code> .
GET	/trace	Provides basic trace information (timestamp, headers, and so on) for HTTP requests.

To enable the Actuator endpoints, all you must do is add the Actuator starter to your build. In a Gradle build specification, that dependency looks like this:

```
compile 'org.springframework.boot:spring-boot-starter-actuator'
```

For a Maven build, the required dependency is as follows:

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

Or, if you're using the Spring Boot CLI, the following `@Grab` should do the trick:

```
@Grab('spring-boot-starter-actuator')
```

No matter which technique you use to add the Actuator to your build, auto-configuration will kick in when the application is running and you enable the Actuator.

The endpoints in table 7.1 can be organized into three distinct categories: configuration endpoints, metrics endpoints, and miscellaneous endpoints. Let's take a look at each of these endpoints, starting with the endpoints that provide insight into the configuration of your application.

7.1.1 **Viewing configuration details**

One of the most common complaints lodged against Spring component-scanning and autowiring is that it's hard to see how all of the components in an application are wired together. Spring Boot auto-configuration makes this problem even worse, as there's even less Spring configuration. At least with explicit configuration, you could look at the XML file or the configuration class and get an idea of the relationships between the beans in the Spring application context.

Personally, I've never had this concern. Maybe it's because I realize that before Spring came along there wasn't any map of the components in my applications.

Nevertheless, if it concerns you that auto-configuration hides how beans are wired up in the Spring application context, then I have some good news! The Actuator has endpoints that give you that missing application component map as well as some insight into the decisions that auto-configuration made when populating the Spring application context.

GETTING A BEAN WIRING REPORT

The most essential endpoint for exploring an application's Spring context is the `/beans` endpoint. This endpoint returns a JSON document describing every single bean in the application context, its Java type, and any of the other beans it's injected with. By performing a GET request to `/beans` (<http://localhost:8080/beans> when running locally), you'll be given information similar to what's shown in the following listing.

Listing 7.1 The `/beans` endpoint exposes the beans in the Spring application context

```
[
  {
    "beans": [
      {
```

```

    "bean": "application",      ← Bean ID
    "dependencies": [],
    "resource": "null",
    "scope": "singleton",      ← Resource file
    "type": "readinglist.Application$$EnhancerBySpringCGLIB$$f363c202"
  },
  {
    "bean": "amazonProperties",
    "dependencies": [],
    "resource": "URL [jar:file:../../readinglist-0.0.1-SNAPSHOT.jar!
                                                             /readinglist/AmazonProperties.class]", ←
    "scope": "singleton",
    "type": "readinglist.AmazonProperties"                      Dependencies
  },
  {
    "bean": "readingListController",
    "dependencies": [      ← Bean scope
      "readingListRepository",
      "amazonProperties"
    ],
    "resource": "URL [jar:file:../../readinglist-0.0.1-SNAPSHOT.jar!
                                                             /readinglist/ReadingListController.class]",
    "scope": "singleton",
    "type": "readinglist.ReadingListController"
  },
  {
    "bean": "readerRepository",
    "dependencies": [
      "(inner bean)#219df4f5",
      "(inner bean)#2c0e7419",
      "(inner bean)#7d86037b",
      "jpaMappingContext"
    ],
    "resource": "null",
    "scope": "singleton",
    "type": "readinglist.ReaderRepository"      ← Java type
  },
  {
    "bean": "readingListRepository",
    "dependencies": [
      "(inner bean)#98ce66",
      "(inner bean)#1fd7add0",
      "(inner bean)#59faabb2",
      "jpaMappingContext"
    ],
    "resource": "null",
    "scope": "singleton",
    "type": "readinglist.ReadingListRepository"
  },
  ...
],
"context": "application",
"parent": null
}
]

```

Listing 7.1 is an abridged listing of the beans from the reading-list application. As you can see, all of the bean entries carry five pieces of information about the bean:

- **bean**—The name or ID of the bean in the Spring application context
- **resource**—The location of the physical .class file (often a URL into the built JAR file, but this might vary depending on how the application is built and run)
- **dependencies**—A list of bean IDs that this bean is injected with
- **scope**—The bean's scope (usually singleton, as that is the default scope)
- **type**—The bean's Java type

Although the beans report doesn't draw a specific picture of how the beans are wired together (for example, via properties or constructor arguments), it does help you visualize the relationships of the beans in the application context. Indeed, it would be reasonably easy to write a utility that processes the beans report and produces a graphical representation of the bean relationships. Be aware, however, that the full bean report includes many beans, including many auto-configured beans, so such a graphic could be quite busy.

EXPLAINING AUTO-CONFIGURATION

Whereas the `/beans` endpoint produces a report telling you what beans are in the Spring application context, the `/autoconfig` endpoint might help you figure out why they're there—or not there.

As mentioned in chapter 2, Spring Boot auto-configuration is built upon Spring conditional configuration. It provides several configuration classes with `@Conditional` annotations referencing conditions that decide whether or not beans should be automatically configured. The `/autoconfig` endpoint provides a report of all the conditions that are evaluated, grouping them by which conditions passed and which failed.

Listing 7.2 shows an excerpt from the auto-configuration report produced for the reading-list application with one passing and one failing condition.

Listing 7.2 An auto-configuration report for the reading-list app

```
{
  "positiveMatches": {                                ← Successful conditions
    ...
    "DataSourceAutoConfiguration.JdbcTemplateConfiguration
                                                #jdbcTemplate": [
      {
        "condition": "OnBeanCondition",
        "message": "@ConditionalOnMissingBean (types:
                    org.springframework.jdbc.core.JdbcOperations;
                    SearchStrategy: all) found no beans"
      }
    ],
    ...
  },
  "negativeMatches": {                                ← Failed conditions
    "ActiveMQAutoConfiguration": [
```

```

    {
      "condition": "OnClassCondition",
      "message": "required @ConditionalOnClass classes not found:
        javax.jms.ConnectionFactory,org.apache.activemq
        .ActiveMQConnectionFactory"
    }
  ],
  ...
}
}

```

In the `positiveMatches` section, you'll find a condition used to decide whether or not Spring Boot should auto-configure a `JdbcTemplate` bean. The match is named `DataSourceAutoConfiguration.JdbcTemplateConfiguration#jdbcTemplate`, which indicates the specific configuration class where this condition is applied. The type of condition is an `OnBeanCondition`, which means that the condition's outcome is determined by the presence or absence of a bean. In this case, the `message` property makes it clear that the condition checks for the absence of a bean of type `JdbcOperations` (the interface that `JdbcTemplate` implements). If no such bean has already been configured, then this condition passes and a `JdbcTemplate` bean will be created.

Similarly, under `negativeMatches`, there's a condition that decides whether or not to configure an `ActiveMQ`. This decision is an `OnClassCondition`, and it hinges on the presence of `ActiveMQConnectionFactory` in the classpath. Because `ActiveMQConnectionFactory` isn't in the classpath, the condition fails and `ActiveMQ` will not be auto-configured.

INSPECTING CONFIGURATION PROPERTIES

In addition to knowing how your application beans are wired together, you might also be interested in learning what environment properties are available and what configuration properties were injected on the beans.

The `/env` endpoint produces a list of all of the environment properties available to the application, whether they're being used or not. This includes environment variables, JVM properties, command-line parameters, and any properties provided in an `application.properties` or `application.yml` file.

The following listing shows an abridged example of what you might get from the `/env` endpoint.

Listing 7.3 The `/env` endpoint reports all properties available

```

{
  "applicationConfig: [classpath:/application.yml]": {
    "amazon.associate_id": "habuma-20",
    "error.whitelabel.enabled": false,
    "logging.level.root": "INFO"
  },
  "profiles": [],
  "servletContextInitParams": {},
  "systemEnvironment": {

```


← Application properties

← Environment variables

```

"BOOK_HOME": "/Users/habuma/Projects/BookProjects/walls6",
"GRADLE_HOME": "/Users/habuma/.sdkman/gradle/current",
"GRAILS_HOME": "/Users/habuma/.sdkman/grails/current",
"GROOVY_HOME": "/Users/habuma/.sdkman/groovy/current",
...
},
"systemProperties": {
  "PID": "682",
  "file.encoding": "UTF-8",
  "file.encoding.pkg": "sun.io",
  "file.separator": "/",
  ...
}
}

```



JVM system properties

Essentially, any property source that can provide properties to a Spring Boot application will be listed in the results of the `/env` endpoint along with the properties provided by that endpoint.

In listing 7.3, properties come from application configuration (`application.yml`), Spring profiles, servlet context initialization parameters, the system environment, and JVM system properties. (In this case, there are no profiles or servlet context initialization parameters.)

It's common to use properties to carry sensitive information such as database or API passwords. To keep that kind of information from being exposed by the `/env` endpoint, any property named (or whose last segment is) “password”, “secret”, or “key” will be rendered as “” in the response from `/env`. For example, if there's a property named “database.password”, it will be rendered in the `/env` response like this:

```
"database.password": ""
```

The `/env` endpoint can also be used to request the value of a single property. Just append the property name to `/env` when making the request. For example, requesting `/env/amazon.associate_id` will yield a response of “habuma-20” (in plain text) when requested against the reading-list application.


As you'll recall from chapter 3, these environment properties come in handy when using the `@ConfigurationProperties` annotation. Beans annotated with `@ConfigurationProperties` can have their instance properties injected with values from the environment. The `/configprops` endpoint produces a report of how those properties are set, whether from injection or otherwise. Listing 7.4 shows an excerpt from the configuration properties report for the reading-list application.

Listing 7.4 A configuration properties report

```

{
  "amazonProperties": {
    "prefix": "amazon",
    "properties": {
      "associateId": "habuma-20"
    }
  }
}

```



Amazon configuration

```

    }
  },
  ...
  "serverProperties": {
    "prefix": "server",
    "properties": {
      "address": null,
      "contextPath": null,
      "port": null,
      "servletPath": "/",
      "sessionTimeout": null,
      "ssl": null,
      "tomcat": {
        "accessLogEnabled": false,
        "accessLogPattern": null,
        "backgroundProcessorDelay": 30,
        "basedir": null,
        "compressableMimeTypes": "text/html,text/xml,text/plain",
        "compression": "off",
        "maxHttpHeaderSize": 0,
        "maxThreads": 0,
        "portHeader": null,
        "protocolHeader": null,
        "remoteIpHeader": null,
        "uriEncoding": null
      }
    },
    ...
  }
},
...
}

```


**Server
configuration**

The first item in this excerpt is the `amazonProperties` bean we created in chapter 3. This report tells us that it's annotated with `@ConfigurationProperties` to have a prefix of "amazon". And it shows that the `associateId` property is set to "habuma-20". This is because in `application.yml`, we set the `amazon.associateId` property to "habuma-20".

You can also see an entry for `serverProperties`—it has a prefix of "server" and several properties that we can work with. Here they all have default values, but you can change any of them by setting a property prefixed with "server". For example, you could change the port that the server listens on by setting the `server.port` property.

Aside from giving insight into how configuration properties are set in the running application, this report is also useful as a quick reference showing all of the properties that you *could* set. For example, if you weren't sure how to set the maximum number of threads in the embedded Tomcat server, a quick look at the configuration properties report would give you a clue that `server.tomcat.maxThreads` is the property you're looking to set.

PRODUCING ENDPOINT-TO-CONTROLLER MAP

When an application is relatively small, it's usually easy to know how all of its controllers are mapped to endpoints. But once the web interface exceeds more than a handful of

controllers and request-handling methods, it might be helpful to have a list of all of the endpoints exposed by the application.

The `/mappings` endpoint provides such a list. Listing 7.5 shows an excerpt of the mappings report from the reading-list application.

Listing 7.5 The controller/endpoint mappings for the reading-list app

```
{
  ...
  " { [/],methods=[GET],params=[],headers=[],consumes=[],produces=[],
                                     custom=[] } ": {
    "bean": "requestMappingHandlerMapping",
    "method": "public java.lang.String readinglist.ReadingListController.
               readersBooks(readinglist.Reader,org.springframework.ui.Model) "
  },
  " { [/],methods=[POST],params=[],headers=[],consumes=[],produces=[],
                                     custom=[] } ": {
    "bean": "requestMappingHandlerMapping",
    "method": "public java.lang.String readinglist.ReadingListController
               .addToReadingList(readinglist.Reader,readinglist.
               Book) "
  },
  " { [/autoconfig],methods=[GET],params=[],headers=[],consumes=[],
                                     produces=[],custom=[] } ": {
    "bean": "endpointHandlerMapping",
    "method": "public java.lang.Object org.springframework.boot
               .actuate.endpoint.mvc.EndpointMvcAdapter.invoke() "
  },
  ...
}
```

**ReadingListController
mappings**

**Auto-configuration
report mapping**

Here we see a handful of endpoint mappings. The key for each mapping is a string containing what appears to be the attributes of Spring MVC's `@RequestMapping` annotation. Indeed, this string gives you a good idea of how the controller is mapped, even if you haven't seen the source code. The value of each mapping has two properties: `bean` and `method`. The `bean` property identifies the name of the Spring bean that the mapping comes from. The `method` property gives the fully qualified method signature of the method for which the mapping is being reported.

The first two mappings are for the request-handling methods in our application's `ReadingListController`. The first shows that an HTTP GET request for the root path (`/`) will be handled by the `readersBooks()` method. The second shows that a POST request is mapped to the `addToReadingList()` method.

The next mapping is for an Actuator-provided endpoint. An HTTP GET request for the `/autoconfig` endpoint will be handled by the `invoke()` method of Spring Boot's `EndpointMvcAdapter` class. There are, of course, many other Actuator endpoints that aren't shown in listing 7.5, but those were omitted from the listing for brevity's sake.

The Actuator's configuration endpoints are great for seeing how your application is configured. But it's also interesting and useful to see what's actually happening within your application while it's running. The metrics endpoints help give a snapshot into an application's runtime internals.

7.1.2 Tapping runtime metrics

When you go to the doctor for a physical exam, the doctor performs a battery of tests to see how your body is performing. Some of them, such as determining your blood type, are important but will not change over time. These kinds of tests give the doctor insight into how your body is configured. Other tests give the doctor a snapshot into how your body is performing during the visit. Your heart rate, blood pressure, and cholesterol level are useful in helping the doctor evaluate your health. These metrics are temporal and likely to change over time, but they're still helpful runtime metrics.

Similarly, taking a snapshot of the runtime metrics is helpful in evaluating the health of an application. The Actuator offers a handful of endpoints that enable you to perform a quick checkup on your application while it's running. Let's take a look at them, starting with the `/metrics` endpoint.

VIEWING APPLICATION METRICS

There are a lot of interesting and useful bits of information about any running application. Knowing the application's memory circumstances (available vs. free), for instance, might help you decide if you need to give the JVM more or less memory to work with. For a web application, it can be helpful knowing at a glance, without scouring web server log files, if there are any requests that are failing or taking too long to serve.

The `/metrics` endpoint provides a snapshot of various counters and gauges in a running application. The following listing shows a sample of what the `/metrics` endpoint might give you.

Listing 7.6 The metrics endpoint provides several useful pieces of runtime data

```
{
  mem: 198144,
  mem.free: 144029,
  processors: 8,
  uptime: 1887794,
  instance.uptime: 1871237,
  systemload.average: 1.33251953125,
  heap.committed: 198144,
  heap.init: 131072,
  heap.used: 54114,
  heap: 1864192,
  threads.peak: 21,
  threads.daemon: 19,
  threads: 21,
  classes: 9749,
  classes.loaded: 9749,
  classes.unloaded: 0,
```

```

gc.ps_scavenge.count: 22,
gc.ps_scavenge.time: 122,
gc.ps_marksweep.count: 2,
gc.ps_marksweep.time: 156,
httpsessions.max: -1,
httpsessions.active: 1,
datasource.primary.active: 0,
datasource.primary.usage: 0,
counter.status.200.beans: 1,
counter.status.200.env: 1,
counter.status.200.login: 3,
counter.status.200.metrics: 2,
counter.status.200.root: 6,
counter.status.200.star-star: 9,
counter.status.302.login: 3,
counter.status.302.logout: 1,
counter.status.302.root: 5,
gauge.response.beans: 169,
gauge.response.env: 165,
gauge.response.login: 3,
gauge.response.logout: 0,
gauge.response.metrics: 2,
gauge.response.root: 11,
gauge.response.star-star: 2
}

```

As you can see, a lot of information is provided by the `/metrics` endpoint. Rather than examine these metrics line by line, which would be tedious, table 7.2 groups them into categories by the type of information they offer.

Table 7.2 Gauges and counters reported by the `/metrics` endpoint

Category	Prefix	What it reports
Garbage collector	<code>gc.*</code>	The count of garbage collections that have occurred and the elapsed garbage collection time for both the mark-sweep and scavenge garbage collectors (from <code>java.lang.management.GarbageCollectorMXBean</code>)
Memory	<code>mem.*</code>	The amount of memory allotted to the application and the amount of memory that is free (from <code>java.lang.Runtime</code>)
Heap	<code>heap.*</code>	The current memory usage (from <code>java.lang.management.MemoryUsage</code>)
Class loader	<code>classes.*</code>	The number of classes that have been loaded and unloaded by the JVM class loader (from <code>java.lang.management.ClassLoadingMXBean</code>)
System	<code>processors</code> <code>uptime</code> <code>instance.uptime</code> <code>systemload.average</code>	System information such as the number of processors (from <code>java.lang.Runtime</code>), uptime (from <code>java.lang.management.RuntimeMXBean</code>), and average system load (from <code>java.lang.management.OperatingSystemMXBean</code>)

Table 7.2 Gauges and counters reported by the `/metrics` endpoint (continued)

Category	Prefix	What it reports
Thread pool	<code>threads.*</code>	The number of threads, daemon threads, and the peak count of threads since the JVM started (from <code>java.lang.management.ThreadMXBean</code>)
Data source	<code>datasource.*</code>	The number of data source connections (from the data source's metadata and only available if there are one or more <code>DataSource</code> beans in the Spring application context)
Tomcat sessions	<code>httpsessions.*</code>	The active and maximum number of sessions in Tomcat (from the embedded Tomcat bean and only available if the application is served via an embedded Tomcat server)
HTTP	<code>counter.status.*</code> <code>gauge.response.*</code>	Various gauges and counters for HTTP requests that the application has served

Notice that some of these metrics, such as the data source and Tomcat session metrics, are only available if the necessary components are in play in the running application. You can also register your own custom application metrics, as you'll see in section 7.4.3.

The HTTP counters and gauges demand a bit more explanation. The number following the `counter.status` prefix is the HTTP status code. What follows that is the path requested. For instance, the metric named `counter.status.200.metrics` indicates the number of times that the `/metrics` endpoint was served with an HTTP status of 200 (OK).

The HTTP gauges are similarly structured but report a different kind of metrics. They're all prefixed with `gauge.response`, indicating that they are gauges for HTTP responses. Following that prefix is the path that the gauge refers to. The value of the metric indicates the time in milliseconds that it took to serve that path the most recent time it was served. For instance, the `gauge.response.beans` metric in table 7.6 indicates that it took 169 milliseconds to serve that request the last time it was served.

You'll notice that there are a few special cases for the counter and gauge paths. The root path refers to the root path or `/`. And `star-star` is a catchall that refers to any path that Spring determines is a static resource, including images, JavaScript, and stylesheets. It also includes any resource that can't be found, which is why you'll often see a `counter.status.404.star-star` metric indicating the count of requests that were met with HTTP 404 (NOT FOUND) status.

Whereas the `/metrics` endpoint fetches a full set of all available metrics, you may only be interested in a single metric. To fetch only one metric value, append the metric's key to the URL path when making the request. For example, to fetch only the amount of free memory, perform a GET request for `/metrics/mem.free`:

```
$ curl localhost:8080/metrics/mem.free
144029
```

It may be useful to know that even though the result from `/metrics/{name}` appears to be plain text, the Content-Type header in the response is set to “application/json; charset=UTF-8”. Therefore, it can be processed as JSON if you need to do so.

TRACING WEB REQUESTS

Although the `/metrics` endpoint gives you some basic counters and timers for web requests, those metrics lack any details. Sometimes it can be helpful, especially when debugging, to know more about the requests that were handled. That’s where the `/trace` endpoint can be handy.

The `/trace` endpoint reports details of all web requests, including details such as the request method, path, timestamp, and request and response headers. Listing 7.7 shows an excerpt of the `/trace` endpoint’s output containing a single request trace entry.

Listing 7.7 The `/trace` endpoint records web request details

```
[
  ...
  {
    "timestamp": 1426378239775,
    "info": {
      "method": "GET",
      "path": "/metrics",
      "headers": {
        "request": {
          "accept": "*/*",
          "host": "localhost:8080",
          "user-agent": "curl/7.37.1"
        },
        "response": {
          "X-Content-Type-Options": "nosniff",
          "X-XSS-Protection": "1; mode=block",
          "Cache-Control":
            "no-cache, no-store, max-age=0, must-revalidate",
          "Pragma": "no-cache",
          "Expires": "0",
          "X-Frame-Options": "DENY",
          "X-Application-Context": "application",
          "Content-Type": "application/json; charset=UTF-8",
          "Transfer-Encoding": "chunked",
          "Date": "Sun, 15 Mar 2015 00:10:39 GMT",
          "status": "200"
        }
      }
    }
  }
]
```

As indicated by the method and path properties, you can see that this trace entry is for a `/metrics` request. The timestamp property (as well as the Date header in the

response) tells you when the request was handled. The `headers` property carries header details for both the request and the response.

Although listing 7.7 only shows a single trace entry, the `/trace` endpoint will report trace details for the 100 most recent requests, including requests for the `/trace` endpoint itself. It maintains the trace data in an in-memory trace repository. Later, in section 7.4.4, you'll see how to create a custom trace repository implementation for a more permanent tracing of requests.

DUMPING THREAD ACTIVITY

In addition to request tracing, thread activity can also be useful in determining what's going on in a running application. The `/dump` endpoint produces a snapshot of current thread activity.

Listing 7.8 The `/dump` endpoint provides a snapshot of an application's threads

```
[
  {
    "threadName": "container-0",
    "threadId": 19,
    "blockedTime": -1,
    "blockedCount": 0,
    "waitedTime": -1,
    "waitedCount": 64,
    "lockName": null,
    "lockOwnerId": -1,
    "lockOwnerName": null,
    "inNative": false,
    "suspended": false,
    "threadState": "TIMED_WAITING",
    "stackTrace": [
      {
        "className": "java.lang.Thread",
        "fileName": "Thread.java",
        "lineNumber": -2,
        "methodName": "sleep",
        "nativeMethod": true
      },
      {
        "className": "org.apache.catalina.core.StandardServer",
        "fileName": "StandardServer.java",
        "lineNumber": 407,
        "methodName": "await",
        "nativeMethod": false
      },
      {
        "className": "org.springframework.boot.context.embedded.
          tomcat.TomcatEmbeddedServletContainer$1",
        "fileName": "TomcatEmbeddedServletContainer.java",
        "lineNumber": 139,
        "methodName": "run",
        "nativeMethod": false
      }
    ]
  }
]
```

```

    ],
    "lockedMonitors": [],
    "lockedSynchronizers": [],
    "lockInfo": null
  },
  ...
]

```

The complete thread dump report includes every thread in the running application. To save space, listing 7.8 shows an abridged entry for a single thread. As you can see, it includes details regarding the blocking and locking status of the thread, among other thread specifics. There's also a stack trace that, in this case, indicates the thread is a Tomcat container thread.

MONITORING APPLICATION HEALTH

If you're ever wondering if your application is up and running or not, you can easily find out by requesting the `/health` endpoint. In the simplest case, the `/health` endpoint reports a simple JSON structure like this:

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

The `status` property reports that the application is up. Of course it is. It doesn't really matter what the response is; any response at all is an indication that the application is running. But the `/health` endpoint has more information than a simple "UP" status.

Some of the information offered by the `/health` endpoint can be sensitive, so unauthenticated requests are only given the simple health status response. If the request is authenticated (for example, if you're logged in), more health information is exposed. Here's some sample health information reported for the reading-list application:

```

{
  "status": "UP",
  "diskSpace": {
    "status": "UP",
    "free": 377423302656,
    "threshold": 10485760
  },
  "db": {
    "status": "UP",
    "database": "H2",
    "hello": 1
  }
}

```

Along with the basic health status, you're also given information regarding the amount of available disk space and the status of the database that the application is using.

All of the information reported by the `/health` endpoints is provided by one or more health indicators, including those listed in table 7.3, that come with Spring Boot.

Table 7.3 Spring Boot's out-of-the-box health indicators

Health indicator	Key	Reports
ApplicationHealthIndicator	none	Always "UP"
DataSourceHealthIndicator	db	"UP" and database type if the database can be contacted; "DOWN" status otherwise
DiskSpaceHealthIndicator	diskSpace	"UP" and available disk space, and "UP" if available space is above a threshold; "DOWN" if there isn't enough disk space
JmsHealthIndicator	jms	"UP" and JMS provider name if the message broker can be contacted; "DOWN" otherwise
MailHealthIndicator	mail	"UP" and the mail server host and port if the mail server can be contacted; "DOWN" otherwise
MongoHealthIndicator	mongo	"UP" and the MongoDB server version; "DOWN" otherwise
RabbitHealthIndicator	rabbit	"UP" and the RabbitMQ broker version; "DOWN" otherwise
RedisHealthIndicator	redis	"UP" and the Redis server version; "DOWN" otherwise
SolrHealthIndicator	solr	"UP" if the Solr server can be contacted; "DOWN" otherwise

These health indicators will be automatically configured as needed. For example, `DataSourceHealthIndicator` will be automatically configured if `javax.sql.DataSource` is available in the classpath. `ApplicationHealthIndicator` and `DiskSpaceHealthIndicator` will always be configured.

In addition to these out-of-the-box health indicators, you'll see how to create custom health indicators in section 7.4.5.

7.1.3 Shutting down the application

Suppose you need to kill your running application. In a microservice architecture, for instance, you might have multiple instances of a microservice application running in the cloud. If one of those instances starts misbehaving, you might decide to shut it down and let the cloud provider restart the failed application for you. In that scenario, the Actuator's `/shutdown` endpoint will prove useful.

In order to shut down your application, you can send a POST request to `/shutdown`. For example, you can shut down your application using the `curl` command-line tool like this:

```
$ curl -X POST http://localhost:8080/shutdown
```


Obviously, the ability to shut down a running application is a dangerous thing, so it's disabled by default. Unless you've explicitly enabled it, you'll get the following response from the POST request:

```
{"message":"This endpoint is disabled"}
```

To enable the `/shutdown` endpoint, configure the `endpoints.shutdown.enabled` property to `true`. For example, add the following lines to `application.yml` to enable the `/shutdown` endpoint:

```
endpoints:
  shutdown:
    enabled: true
```

Once the `/shutdown` endpoint is enabled, you want to make sure that not just anybody can kill your application. You should secure the `/shutdown` endpoint, requiring that only authorized users are allowed to bring the application down. You'll see how to secure Actuator endpoints in section 7.5.

7.1.4 *Fetching application information*

Spring Boot's Actuator has one more endpoint you might find useful. The `/info` endpoint reports any information about your application that you might want to expose to callers. The default response to a GET request to `/info` looks like this:

```
{}
```

Obviously, an empty JSON object isn't very useful. But you can add any information to the `/info` endpoint's response by simply configuring properties prefixed with `info`. For example, suppose you want to provide a contact email in the `/info` endpoint response. You could set a property named `info.contactEmail` like this in `application.yml`:

```
info:
  contactEmail: support@myreadinglist.com
```

Now if you request the `/info` endpoint, you'll get the following response:

```
{
  "contactEmail":"support@myreadinglist.com"
}
```

Properties in the `/info` response can also be nested. For example, suppose that you want to provide both a support email and a support phone number. In `application.yml`, you might configure the following properties:

```
info:
  contact:
    email: support@myreadinglist.com
    phone: 1-888-555-1971
```

The JSON returned from the `/info` endpoint will include a `contact` property that itself has `email` and `phone` properties:

```
{
  "contact": {
    "email": "support@myreadinglist.com",
    "phone": "1-888-555-1971"
  }
}
```

Adding properties to the `/info` endpoint is just one of many ways you can customize Actuator behavior. Later in section 7.4, we'll look at other ways that you can configure and extend the Actuator. But first, let's see how to secure the Actuator's endpoints.

7.2 Connecting to the Actuator remote shell

You've seen how the Actuator provides some very useful information over REST endpoints. An optional way to dig into the internals of a running application is by way of a remote shell. Spring Boot integrates with CRaSH, a shell that can be embedded into any Java application. Spring Boot also extends CRaSH with a handful of Spring Boot-specific commands that offer much of the same functionality as the Actuator's endpoints.

In order to use the remote shell, you'll need to add the remote shell starter as a dependency. The Gradle dependency you'll need looks like this:

```
compile("org.springframework.boot:spring-boot-starter-remote-shell")
```

If you're building your project with Maven, you'll need the following dependency in your `pom.xml` file:

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

And if you're developing an application to run with the Spring Boot CLI, the following `@Grab` is what you'll need:

```
@Grab("spring-boot-starter-remote-shell")
```

With the remote shell added as a dependency, you can now build and run the application. As it's starting up, watch for the password to be written to the log in a line that looks something like this:

```
Using default security password: efe30c70-5bf0-43b1-9d50-c7a02dda7d79
```

The username that goes with that password is "user". The password itself is randomly generated and will be different each time you run the application.


```

Terminal - ssh - 140x40
ssh

> autoconfig
Endpoint: autoConfigurationAuditEndpoint

Positive Matches:
=====
AuditAutoConfiguration#authenticationAuditListener:
  OnClassCondition: @ConditionalOnClass classes found: org.springframework.security.authentication.event.AbstractAuthenticationEvent
AuditAutoConfiguration#authorizationAuditListener:
  OnClassCondition: @ConditionalOnClass classes found: org.springframework.security.access.event.AbstractAuthorizationEvent
AuditAutoConfiguration#auditEventRepositoryConfiguration:
  OnBeanCondition: @ConditionalOnMissingBean (types: org.springframework.boot.actuate.audit.AuditEventRepository; SearchStrategy: all) found no beans
CrshAutoConfiguration:
  OnClassCondition: @ConditionalOnClass classes found: org.crsh.plugin.PluginLifeCycle
CrshAutoConfiguration#shellBootstrap:
  OnBeanCondition: @ConditionalOnMissingBean (types: org.crsh.plugin.PluginLifeCycle; SearchStrategy: all) found no beans
CrshAutoConfiguration#authenticationManagerAdapterAutoConfiguration:
  OnPropertyCondition: matched
  OnBeanCondition: @ConditionalOnBean (types: org.springframework.security.authentication.AuthenticationManager; SearchStrategy: all) found the following [authenticationManager]
CrshAutoConfiguration#authenticationManagerAdapterAutoConfiguration#springAuthenticationProperties:
  OnBeanCondition: @ConditionalOnMissingBean (types: org.springframework.boot.actuate.autoconfigure.ShellProperties$CrshShellAuthenticationP
  OnPropertyCondition: @ConditionalOnMissingBean (types: org.springframework.boot.actuate.autoconfigure.ShellProperties$CrshShellAuthenticationP
  OnBeanCondition: @ConditionalOnMissingBean (names: websocketContainerCustomizer; SearchStrategy: all) found no beans

Negative Matches
=====
CrshAutoConfiguration#jaasAuthenticationProperties:
  OnPropertyCondition: @ConditionalOnProperty missing required properties shell.auth
CrshAutoConfiguration#keyAuthenticationProperties:
  OnPropertyCondition: @ConditionalOnProperty missing required properties shell.auth
CrshAutoConfiguration#implAuthenticationProperties:
  OnPropertyCondition: matched
  OnBeanCondition: @ConditionalOnMissingBean (types: org.springframework.boot.actuate.autoconfigure.ShellProperties$CrshShellAuthenticationP
  properties; SearchStrategy: all) found the following [springAuthenticationProperties]
EndpointMBeanExportAutoConfiguration#mbeanServer:
  OnBeanCondition: @ConditionalOnMissingBean (types: javax.management.MBeanServer; SearchStrategy: all) found the following [mbeanServer]
HealthIndicatorAutoConfiguration#applicationHealthIndicator:
  OnBeanCondition: @ConditionalOnMissingBean (types: org.springframework.boot.actuate.health.HealthIndicator; SearchStrategy: all) found the following [diskSpaceHealthIndicator, dbHealthIndicator]
HealthIndicatorAutoConfiguration#mongoHealthIndicatorConfiguration:
  OnPropertyCondition: matched
  OnBeanCondition: @ConditionalOnBean (types: org.springframework.data.mongodb.core.MongoTemplate; SearchStrategy: all) found no beans
HealthIndicatorAutoConfiguration#rabbitHealthIndicatorConfiguration:
  OnPropertyCondition: matched
  OnBeanCondition: @ConditionalOnBean (types: org.springframework.amqp.rabbit.core.RabbitTemplate; SearchStrategy: all) found no beans
HealthIndicatorAutoConfiguration#redisHealthIndicatorConfiguration:
  OnPropertyCondition: matched
  OnBeanCondition: @ConditionalOnBean (types: org.springframework.data.redis.connection.RedisConnectionFactory; SearchStrategy: all) found n

```

Figure 7.1 Output of autoconfig command

We're not going to dwell on any of the shell commands provided natively by CRaSH, but you might want to consider piping the results of the autoconfig command to CRaSH's less command:

```
> autoconfig | less
```

The less command is much like the same-named command in Unix shells; it enables you to page back and forth through a file. The autoconfig output is lengthy, but piping it to less will make it easier to read and navigate.

7.2.2 Listing application beans

The output from the autoconfig shell command and the /autoconfig endpoint were similar but different. In contrast, you'll find that the results from the beans command are exactly the same as those from the /beans endpoint, as the screenshot in figure 7.2 shows.

It's difficult to demonstrate the live dashboard behavior of the `metrics` command with a static figure in a book. But try to imagine that as memory, heap, and threads are consumed and released and as classes are loaded, the numbers shown in the dashboard will change to reflect the current values.

Once you're finished looking at the metrics offered by the `metrics` command, press Ctrl-C to return to the shell.

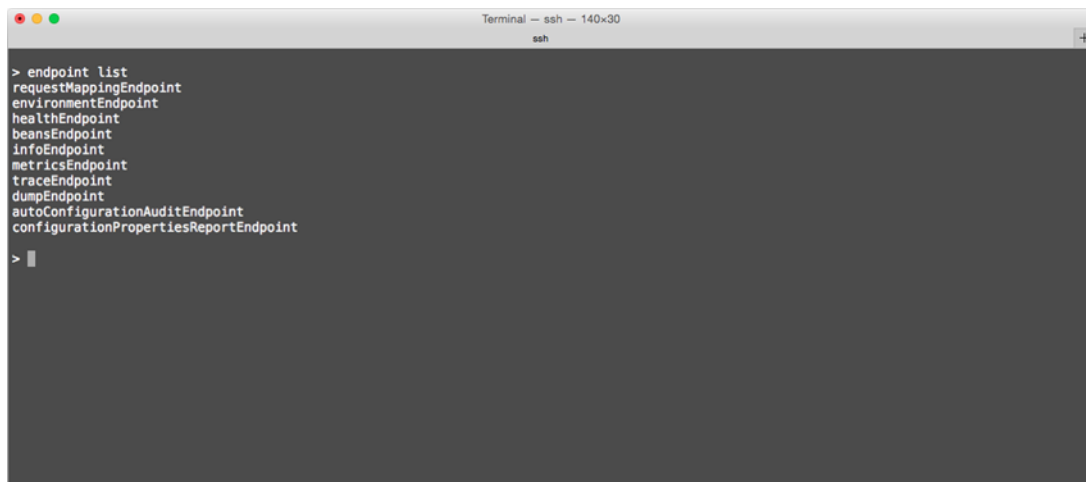
7.2.4 Invoking Actuator endpoints

You've probably realized by now that not all of the Actuator's endpoints have corresponding commands in the shell. Does that mean that the shell can't be a full replacement for the Actuator endpoints? Will you still have to query the endpoints directly for some of the internals offered by the Actuator? Although the shell doesn't pair a command up with each of the endpoints, the `endpoint` command enables you to invoke Actuator endpoints from within the shell.

First, you need to know which endpoint you want to invoke. You can get a list of endpoints by issuing `endpoint list` at the shell prompt, as shown in figure 7.4. Notice that the endpoints listed are referred to by their bean names, not by their URL paths.

When you want to call one of the endpoints from the shell, you'll use the `endpoint invoke` command, giving it the endpoint's bean name without the "Endpoint" suffix. For example, to invoke the health endpoint, you'd issue `endpoint invoke health` at the shell prompt, as shown in figure 7.5.

Notice that the results coming back from the endpoint are in the form of a raw, unformatted JSON document. Although it may be nice to be able to invoke the Actuator's endpoints from within the shell, the results can be a bit difficult to read. Out of the box, there's not much that can be done about that. But if you're feeling adventurous, you can create a custom CRaSH shell command that accepts the unformatted JSON via

A screenshot of a terminal window titled "Terminal - ssh - 140x30" with a "ssh" prompt. The user has entered the command `> endpoint list`, and the terminal displays a list of Actuator endpoints: `requestMappingEndpoint`, `environmentEndpoint`, `healthEndpoint`, `beansEndpoint`, `infoEndpoint`, `metricsEndpoint`, `traceEndpoint`, `dumpEndpoint`, `autoConfigurationAuditEndpoint`, and `configurationPropertiesReportEndpoint`. The prompt `> |` is visible at the bottom of the list.

```
Terminal - ssh - 140x30
ssh
> endpoint list
requestMappingEndpoint
environmentEndpoint
healthEndpoint
beansEndpoint
infoEndpoint
metricsEndpoint
traceEndpoint
dumpEndpoint
autoConfigurationAuditEndpoint
configurationPropertiesReportEndpoint
> |
```

Figure 7.4 Getting a list of endpoints