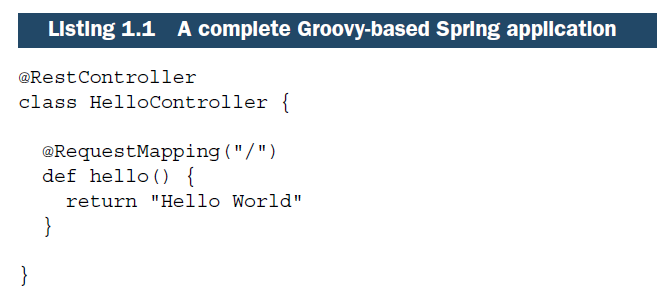
* Spring Boot enables automatic configuration, making it possible for Spring to intelligently detect what kind of application you’re building and automatically configure the components necessary to support the application’s needs.
* Spring Boot starter dependencies make it even easier to select which build-time and runtime libraries to include in your application builds by aggregating commonly needed dependencies. Spring Boot starters not only keep the dependencies section of your build specifications shorter, they keep you from having to think too hard about the specific libraries and versions you need.
* Spring Boot’s command-line interface offers a compelling option for developing Spring applications in Groovy with minimal noise or ceremony common in Java applications
* Spring Boot’s Actuator gives you insight into the inner workings of a running application. You can see exactly what beans are in the Spring application context, how Spring MVC controllers are mapped to paths, the configuration properties available to your application, and much more.

***Spring rebooted***

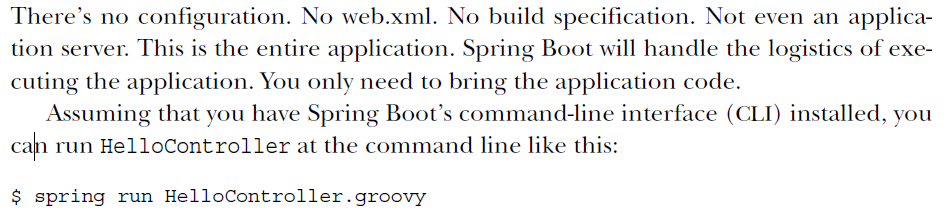
* If you are using spring you know that you have to do some configurations in spring to work with the spring based application.
* For example, Enabling certain Spring features such as transaction management and Spring MVC required explicit configuration, either in XML or Java. Enabling third-party library features such as Thymeleaf-based web views required explicit configuration. Configuring servlets and filters (such as Spring’s DispatcherServlet) required explicit configuration in web.xml or in a servlet initializer. Component-scanning reduced configuration and Java configuration made it less awkward, but Spring still required a lot of configuration.
* All of that configuration represents development friction. Any time spent writing configuration is time spent not writing application logic.

***Taking a fresh look at Spring***

* A project structure, complete with a Maven or Gradle build file including required dependencies. At the very least, you’ll need Spring MVC and the Servlet API expressed as dependencies.
* A web.xml file (or a WebApplicationInitializer implementation) that declares Spring’s DispatcherServlet.
* A Spring configuration that enables Spring MVC.
* A controller class that will respond to HTTP requests with “Hello World”.
* A web application server, such as Tomcat, to deploy the application to.

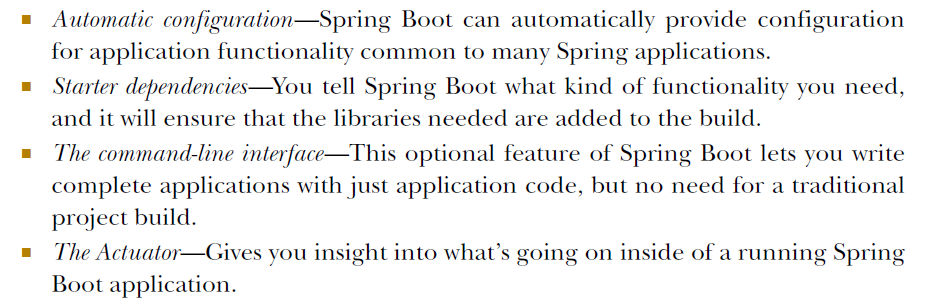
Suppose for a moment that the controller is all you need. As it turns out, the Groovy-based controller class shown in listing 1.1 is a complete (even if simple) Spring application. 

There’s no configuration. No web.xml. No build specification. Not even an application server. This is the entire application. Spring Boot will handle the logistics of executing the application. You only need to bring the application code.



***Examining Spring Boot essentials***

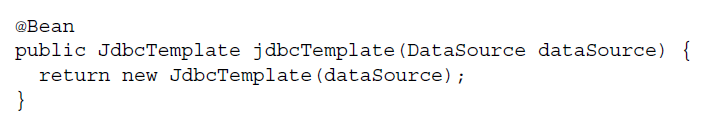
Spring Boot brings a great deal of magic to Spring application development. But there are four core tricks that it performs:

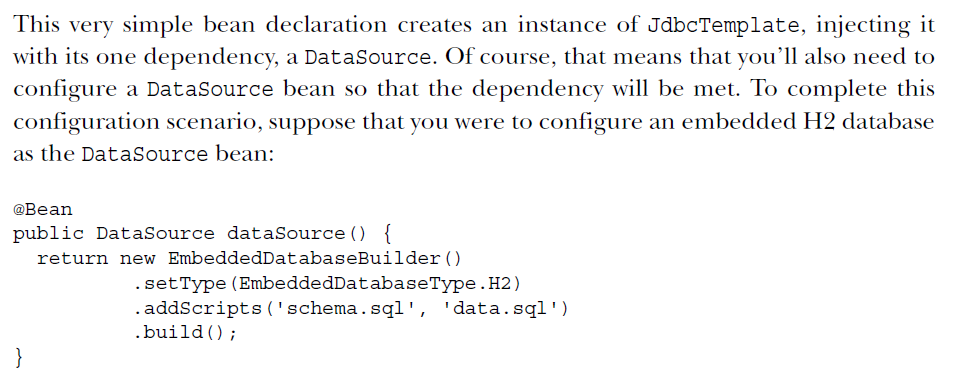


Each of these features serves to simplify Spring application development in its own way.

**AUTO-CONFIGURATION**

In any given Spring application’s source code, you’ll find either Java configuration or XML configuration (or both) that enables certain supporting features and functionality for the application. For example, if you’ve ever written an application that accesses a relational database with JDBC, you’ve probably configured Spring’s JdbcTemplate as a bean in the Spring application context. I’ll bet the configuration looked a lot like this:

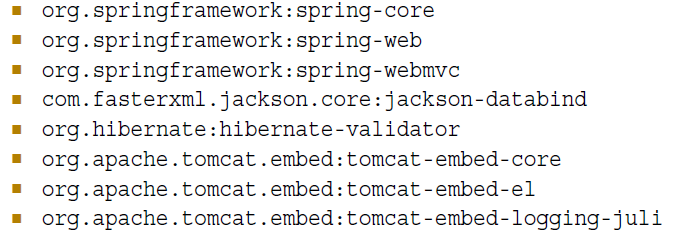




* This bean configuration method creates an embedded database, specifying two SQL scripts to execute on the embedded database. The build() method returns a DataSource that references the embedded database.
* Neither of these two bean configuration methods is terribly complex or lengthy. But they represent just a fraction of the configuration in a typical Spring application. Moreover, there are countless Spring applications that will have these exact same methods. Any application that needs an embedded database and a JdbcTemplate will need those methods. In short, it’s boilerplate configuration.
* If Spring Boot detects that you have the H2 database library in your application’s classpath, it will automatically configure an embedded H2 database. If JdbcTemplate is in the classpath, then it will also configure a JdbcTemplate bean for you. There’s no need for you to worry about configuring those beans. They’ll be configured for you, ready to inject into any of the beans you write.
* There are several dozen ways that Spring Boot can take the burden of configuration off your hands, including auto-configuration for the Java Persistence API (JPA), Thymeleaf templates, security, and Spring MVC. We’ll dive into autoconfiguration later.

**STARTER DEPENDENCIES**

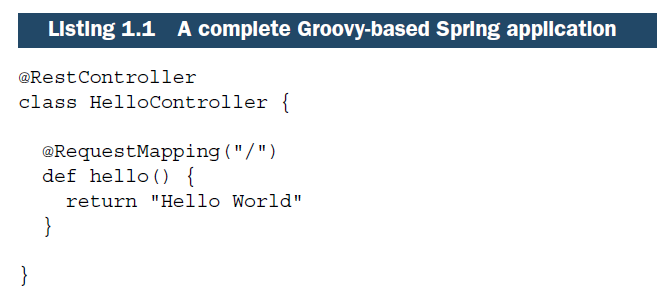
* It can be challenging to add dependencies to a project’s build. What library do you need? What are its group and artifact? Which version do you need? Will that version play well with other dependencies in the same project?
* Starter dependencies are really just special Maven (and Gradle) dependencies that take advantage of transitive dependency resolution to aggregate commonly used libraries under a handful of feature-defined dependencies.
* For example, suppose that you’re going to build a REST API with Spring MVC that works with JSON resource representations. Additionally, you want to apply declarative validation per the JSR-303 specification and serve the application using an embedded Tomcat server. To accomplish all of this, you’ll need (at minimum) the following eight dependencies in your Maven or Gradle build:

****

* On the other hand, if you were to take advantage of Spring Boot starter dependencies, you could simply add the Spring Boot “web” starter (org.springframework.boot:spring-boot-starter-web) as a build dependency. This single dependency will transitively pull in all of those other dependencies so you don’t have to ask for them all.
* But there’s something more subtle about starter dependencies than simply reducing build dependency count. Notice that by adding the “web” starter to your build, you’re specifying a type of functionality that your application needs. Your app is a web application, so you add the “web” starter. Likewise, if your application will use JPA persistence, then you can add the “jpa” starter. If it needs security, you can add the “security” starter. In short, you no longer need to think about what libraries you’ll need to support certain functionality; you simply ask for that functionality by way of the pertinent starter dependency.
* Also note that Spring Boot’s starter dependencies free you from worrying about which versions of these libraries you need. The versions of the libraries that the starters pull in have been tested together so that you can be confident that there will be no incompatibilities between them.

**THE COMMAND-LINE INTERFACE (CLI)**

Spring Boot’s CLI leverages starter dependencies and auto-configuration to let you focus on writing code. Not only that, did you notice that there are no import lines in listing 1.1? How did the CLI know what packages RequestMapping and RestController come from? For that matter, how did those classes end up in the classpath?



The short answer is that the CLI detected that those types are being used, and it knows which starter dependencies to add to the classpath to make it work. Once those dependencies are in the classpath, a series of auto-configuration kicks in and ensures that DispatcherServlet and Spring MVC are enabled so that the controller can respond to HTTP requests.

We will learn more about Spring Boot CLI later.

**THE ACTUATOR**

The Actuator offers the ability to inspect the internals of your application at runtime.

With the Actuator installed, you can inspect the inner workings of your application, including details such as:

* What beans have been configured in the Spring application context
* What decisions were made by Spring Boot’s auto-configuration
* What environment variables, system properties, configuration properties, and command-line arguments are available to your application
* The current state of the threads in and supporting your application
* A trace of recent HTTP requests handled by your application
* Various metrics pertaining to memory usage, garbage collection, web requests, and data source usage

Note:

The Actuator exposes this information in two ways: via web endpoints or via a shell interface. In the latter case, you can actually open a secure shell (SSH) into your application and issue commands to inspect your application as it runs.

***What Spring Boot isn’t***

* Spring Boot is not an application server. This misconception stems from the fact that it’s possible to create web applications as self-executable JAR files that can be run at the command line without deploying applications to a conventional Java application server. Spring Boot accomplishes this by embedding a servlet container (Tomcat, Jetty, or Undertow) within the application. But it’s the embedded servlet container that provides application server functionality, not Spring Boot itself.
* Spring Boot doesn’t implement any enterprise Java specifications such as JPA or JMS. It does support several enterprise Java specifications, but it does so by automatically configuring beans in Spring that support those features. For instance, Spring Boot doesn’t implement JPA, but it does support JPA by auto-configuring the appropriate beans for a JPA implementation (such as Hibernate).
* Spring Boot doesn’t employ any form of code generation to accomplish its magic. Instead, it leverages conditional configuration features from Spring 4, along with transitive dependency resolution offered by Maven and Gradle, to automatically configure beans in the Spring application context.