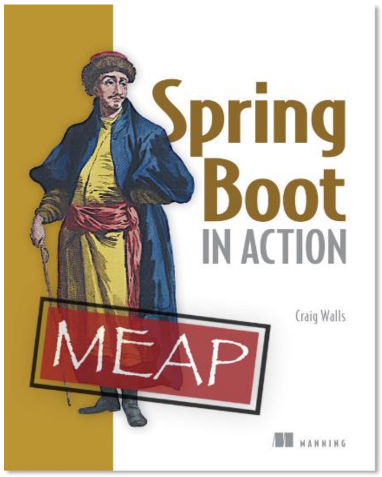
Examining Spring Boot essentials

By Craig Walls, Spring Boot in Action



Spring Boot brings a great deal of magic to Spring application development. But there are four core tricks that it performs. In this article, excerpted from Spring Boot in Action, we introduce those core tricks.

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

- Automatic configuration: Spring Boot can automatically provide configuration for application functionality common to many Spring applications.
- Starter dependencies: You tell Spring Boot what kind of functionality you need and it will ensure that the libraries needed are added to the build.
- The command line interface: This optional feature of Spring Boot lets you write complete applications with just application code, but no need for a traditional project build.
- The Actuator: Gives you insight into what is going on inside of a running Spring Boot application.

Each of these features serves to simplify Spring application development in its own way. Let's take a quick look at what each offers.

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:

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
   return new JdbcTemplate(dataSource);
}
```

This very simple bean declaration creates an instance of `JdbcTemplate`, injecting it with its one dependency, a `DataSource`. Of course, that means that you'll also need to configure a `DataSource` bean so that the dependency will be met. To complete this configuration scenario, suppose that you were to configure an embedded H2 database as the `DataSource` bean:

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
   return new JdbcTemplate(dataSource);
}
@Bean
public DataSource dataSource() {
   return new EmbeddedDatabaseBuilder()
       .setType(EmbeddedDatabaseType.H2)
       .addScripts('schema.sql', 'data.sql')
       .build();
}
```

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 it's so common, then why should you have to write it?

Spring Boot can automatically configure these common configuration scenarios. 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's a lot more to Spring Boot auto-configuration than embedded databases and `JdbcTemplate`. There are several dozen ways that Spring Boot can take the burden of configuration off of your hands, including auto-configuration for the Java Persistence API (JPA), Thymeleaf templates, security, and Spring MVC.

STARTER DEPENDENCIES

It can be challenging to add dependencies to a project's build. What library do you need? What is its group and artifact? Which version do you need? Will that version play well with other dependencies in the same project?

Spring Boot offers help with project dependency management by way of starter dependencies. 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 are going to build a REST API with Spring MVC that works 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 8 dependencies in your Maven or Gradle build:

```
org.springframework:spring-core
org.springframework:spring-web
org.springframework:spring-webmvc
com.fasterxml.jackson.core:jackson-databind
org.hibernate:hibernate-validator
org.apache.tomcat.embed:tomcat-embed-core
org.apache.tomcat.embed:tomcat-embed-el
org.apache.tomcat.embed:tomcat-embed-logging-juli
```

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, then 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)

In addition to auto-configuration and starter dependencies, Spring Boot also offers an intriguing new way to quickly write Spring applications. The Spring Boot CLI makes it possible to write applications by doing more than writing the application code.

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 XREF ex_HelloController_groovy? 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 is enabled so that the controller can respond to HTTP requests.

Spring Boot’s CLI is an optional piece of Spring Boot’s power. Although it provides tremendous power and simplicity for Spring development, it also introduces a rather unconventional development model. If this development model is too extreme for your taste, then no problem. You can still take advantage of everything else that Spring Boot has to offer even if you don’t use the CLI.

THE ACTUATOR

The final piece of the Spring Boot puzzle is the Actuator. Where the other parts of Spring Boot simplify Spring development, the Actuator instead 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

The actuator exposes this information in two ways: via web endpoints and/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.

By now you should have a general idea of what Spring Boot brings to the table.