Chapter 1. Spring Boot in a Nutshell

This chapter explores the three core features of Spring Boot and how they are force multipliers for you as a developer.

Spring Boot's Three Foundational Features

The three core features of Spring Boot upon which everything else builds are simplified dependency management, simplified deployment, and autoconfiguration.

Starters for Simplified Dependency Management

One of the genius aspects of Spring Boot is that it makes dependency management…manageable.

If you've been developing software of any import for any length of time, you've almost certainly had to contend with several headaches surrounding dependency management. Any capability you provide in your application typically requires a number of "frontline" dependencies. For example, if you want to provide a RESTful web API, you must provide a way to expose endpoints over HTTP, listen for requests, tie those endpoints to methods/functions that will process those requests, and then build and return appropriate responses.

Almost invariably, each primary dependency incorporates numerous other secondary dependencies in order to fulfill its promised functionality. Continuing with our example of providing a RESTful API, we might expect to see a collection of dependencies (in some sensible but debatable structure) that includes code to supply responses in a particular format, e.g., JSON, XML, HTML; code to marshal/unmarshal objects to requested format(s); code to listen for and process requests and return responses to same; code to decode complex URIs used to create versatile APIs; code to support various wire protocols; and more.

Even for this fairly simple example, we're already likely to require a large number of dependencies in our build file. And we haven't even considered what functionality we may wish to include in our application at this point, only its outward interactions.

Now, let's talk versions. Of each and every one of those dependencies.

Using libraries together requires a certain degree of rigor, as one version of a particular dependency may have been tested (or even function correctly) only with a specific version of another dependency. When these issues inevitably arise, it leads to what I refer to as "Dependency Whack-a-Mole."

Like its namesake carnival game, Dependency Whack-a-Mole can be a frustrating experience. Unlike its namesake, when it comes to chasing down and bashing bugs stemming from mismatches that pop up between dependencies, there are no prizes, only elusive conclusive diagnoses and *hours* wasted pursuing them.

Enter Spring Boot and its starters. Spring Boot starters are Bills of Materials (BOMs) built around the proven premise that the vast majority of times you provide a particular capability, you do it in nearly the same way, nearly every time.

In the previous example, each time we build an API, we expose endpoints, listen for requests, process requests, convert to and from objects, exchange information in 1+ standard formats, send and receive data over the wire using a particular protocol, and more. This design/development/usage pattern doesn't vary much; it's an approach adopted industry-wide, with few variations. And like other similar patterns, it's handily captured in a Spring Boot starter.

Adding a single starter, e.g., `spring-boot-starter-web`, provides all of those related functionalities in a *single application dependency*. All dependencies encompassed by that single starter are version-synchronized too, meaning that they've been tested successfully together and the included version of library A is proven to function properly with the included version of library B…and C…and D…etc. This dramatically simplifies your dependency list and your life, as it practically eliminates any chance you'll

have difficult-to-identify version conflicts among dependencies you need to provide your application's critical capabilities.

In those rare cases when you must incorporate functionality provided by a different version of an included dependency, you can simply override the tested version.

CAUTION

If you must override the default version of a dependency, do so…but you should probably increase your level of testing to mitigate risks you introduce by doing so.

You can also exclude dependencies if they are unnecessary for your application, but the same cautionary note applies.

All in all, Spring Boot's concept of starters greatly streamlines your dependencies and reduces the work required to add whole sets of capabilities to your applications. It also dramatically diminishes the overhead you incur testing, maintaining, and upgrading them.

Executable JARs for Simplified Deployment

Long ago, in the days when application servers roamed the earth, deployments of Java applications were a complex affair.

In order to field a working application with, say, database access—like many microservices today and nearly all monoliths then and now—you would need to do the following:

1. Install and configure the Application Server.
2. Install database drivers.
3. Create a database connection.
4. Create a connection pool.
5. Build and test your application.
6. Deploy your application and its (usually numerous) dependencies to the Application Server.

Note that this list assumes you had administrators to configure the machine/virtual machine and that at some point you had created the data-

base independently of this process.

Spring Boot turned much of this cumbersome deployment process on its head and collapsed the previous steps into one, or perhaps two, if you count copying or `cf push`-ing a single file to a destination as an actual *step*.

Spring Boot wasn't the origin of the so-called über JAR, but it revolutionized it. Rather than teasing out every file from the application JAR and all dependent JARs, then combining them into a single destination JAR—sometimes referred to as *shading*—the designers of Spring Boot approached things from a truly novel perspective: what if we could *nest JARs*, retaining their intended and delivered format?

Nesting JARs instead of shading them alleviates *many* potential problems, as there are no potential version conflicts to be encountered when dependency JAR A and dependency JAR B each use a different version of C; it also removes potential legal issues due to repackaging software and combining it with other software using a different license. Keeping all dependent JARs in their original format cleanly avoids those and other issues.

It is also trivial to extract the contents of a Spring Boot executable JAR, should you wish to do that. There are some good reasons for doing so in some circumstances, and I'll discuss those in this book as well. For now, just know that the Spring Boot executable JAR has you covered.

That single Spring Boot JAR with all dependencies makes deployment a breeze. Rather than collecting and verifying all dependencies are deployed, the Spring Boot plug-in ensures they're all zipped into the output JAR. Once you have that, the application can be run anywhere there is a Java Virtual Machine (JVM) just by executing a command like `java -jar <SpringBootAppName.jar>`.

There's more.

By setting a single property in your build file, the Spring Boot build plug-in can also make that single JAR entirely (self) executable. Still assuming a JVM is present, rather than having to type or script that entire bothersome line of `java -jar <SpringBootAppName.jar>`, you could simply type `<SpringBootAppName.jar>` (replacing with your filename, of

course), and Bob's your uncle—you're up and running. It doesn't get any easier than that.

Autoconfiguration

Sometimes called "magic" by those new to Spring Boot, autoconfiguration is perhaps the greatest "force multiplier" that Spring Boot brings to developers. I often refer to it as a developer's superpower: Spring Boot gives you *insane productivity* by bringing opinions to widely used and -repeated use cases.

Opinions in software? How does that help?!?

If you've been a developer for very long at all, you'll doubtless have noticed that some patterns repeat themselves frequently. Not perfectly, of course, but in the high percentages; perhaps 80–90% of the time things fall within a certain range of design, development, or activity.

I alluded earlier to this repetition within software, as this is what makes Spring Boot's starters amazingly consistent and useful. The repetition also means that these activities, when it comes to the code that must be written to complete a particular task, are ripe for streamlining.

To borrow an example from Spring Data, a Spring Boot–related and –enabled project, we know that every time we need to access a database, we need to open some manner of connection to that database. We also know that when our application completes its tasks, that connection must be closed to avoid potential issues. In between, we are likely to make numerous requests to the database using queries—simple and complex, read-only and write-enabled—and those queries will require some effort to create properly.

Now imagine we could streamline all of that. Automatically open a connection when we specify the database. Automatically close the connection when the application terminates. Follow a simple and expected convention to create queries *automatically* with minimal effort from you, the developer. Enable easy customization of even that minimal code, again by simple convention, to create complex bespoke queries that are reliably consistent and efficient.

This approach to code is sometimes referred to as *convention over configuration*, and if you're new to a particular convention, it can appear mildly jarring (no pun intended) at first glance. But if you've implemented similar features before, writing often hundreds of repetitive, mind-numbing lines of setup/teardown/configuration code to accomplish even the simplest of tasks, it's like a gust of fresh air. Spring Boot (and most Spring projects) follow the *convention over configuration* mantra, providing the assurance that if you follow simple, well-established and -documented conventions to do something, the configuration code you must write is minimal, or none at all.

Another way in which autoconfiguration gives you superpowers is the Spring team's laserlike focus on "developer-first" environment configuration. As developers, we are most productive when we can focus on the task at hand and not a million setup chores. How does Spring Boot make that happen?

Let's borrow an example from another Spring Boot–related project, Spring Cloud Stream: when connecting to a messaging platform like RabbitMQ or Apache Kafka, a developer typically must specify certain settings for said messaging platform in order to connect to and use it—hostname, port, credentials, and more. Focusing on the development experience means that defaults are provided when none are specified that *favor the developer working locally*: localhost, default port, etc. This makes sense as an *opinion* because it's nearly 100% consistent for development environments, while it isn't so in production. In prod, you would need to provide specific values due to widely varying platform and hosting environments.

Shared development projects using those defaults also eliminate a great deal of time required for developer environment setup. Win for you, win for your team.

There are occasions when your specific use cases don't exactly match the 80–90% of use cases that are typical, when you fall into the other 10–20% of valid use cases. In those instances, autoconfiguration can be selectively overridden, or even disabled entirely, but you lose all of your superpowers then, of course. Overriding certain opinions is typically a matter of setting one or more properties as you wish them to be or providing one or more beans to accomplish something that Spring Boot would normally

autoconfigure on your behalf. In other words, this is often a very simple matter to accomplish on those rare occasions when you must do so. In the end, autoconfiguration is a powerful tool that silently and tirelessly works on your behalf to make your life easier and you insanely productive.

Summary

The three core features of Spring Boot upon which everything else builds are simplified dependency management, simplified deployment, and autoconfiguration. All three are customizable, but you'll seldom need to do so. And all three work hard to make you a better, more productive developer. Spring Boot gives you wings!

In the next chapter, we'll take a look at some of the great options you have when getting started creating Spring Boot applications. Choices are good!