Chapter 2. Choosing Your Tools and Getting Started

Getting started creating Spring Boot apps is easy, as you'll soon see. The most difficult part might be deciding which of the available options you'd like to choose.

In this chapter, we'll examine some of the excellent choices you have available to create Spring Boot applications: build systems, languages, toolchains, code editors, and more.

Maven or Gradle?

Historically, Java application developers have had a few options for project build tools. Some have fallen out of favor over time—for good reason—and now we've coalesced as a community around two: Maven and Gradle. Spring Boot supports both with equal aplomb.

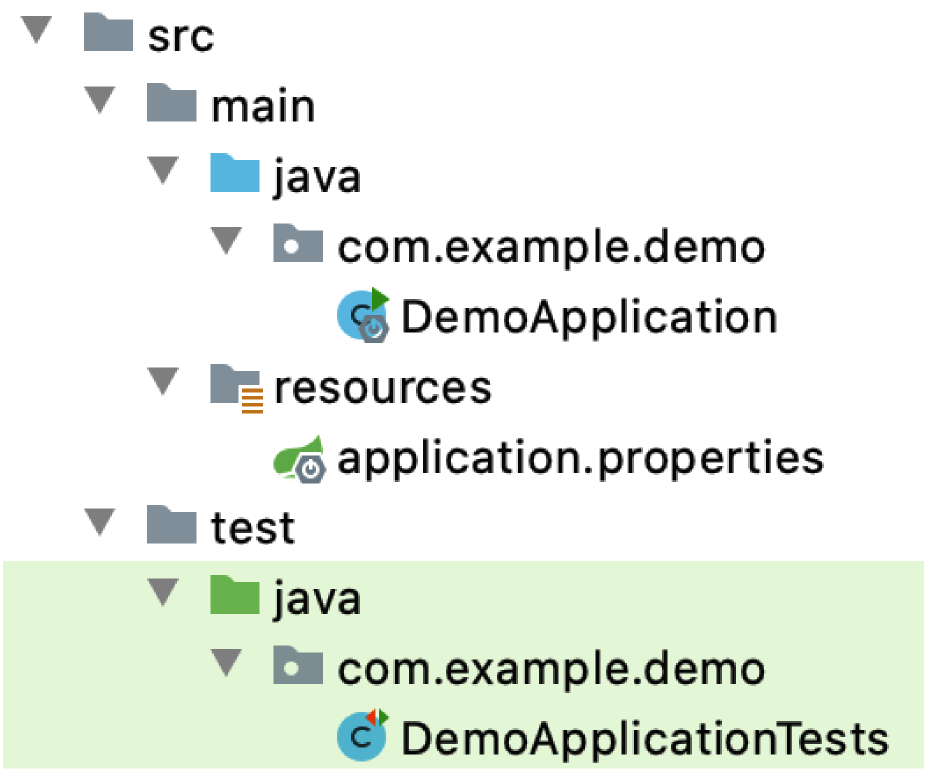
Apache Maven

Maven is a popular and solid choice for a build automation system. It has been around for quite some time, having had its beginning in 2002 and becoming a top-level project at the Apache Software Foundation in 2003. Its declarative approach was (and is) conceptually simpler than the alternatives of the time and of now: simply create an XML-formatted file named *pom.xml* with desired dependencies and plug-ins. When you execute the `mvn` command, you can specify a "phase" to complete, which accomplishes a desired task like compiling, removing prior output(s), packaging, running an application, and more:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.4.0</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
```

```
		<artifactId>demo</artifactId>
		<version>0.0.1-SNAPSHOT</version>
		<name>demo</name>
		<description>Demo project for Spring Boot</description>

		<properties>
			<java.version>11</java.version>
		</properties>

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

			<dependency>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-starter-test</artifactId>
				<scope>test</scope>
			</dependency>
		</dependencies>

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

	</project>
```

Maven also creates and expects a particular project structure by convention. You typically shouldn't deviate much—if at all—from that structure unless you are prepared to fight your build tool, a counterproductive quest if there ever was one. For the vast majority of projects, the conventional Maven structure works perfectly, so it isn't something you'll likely need to change. Figure 2-1 shows a Spring Boot application with the typical Maven project structure.



```
src
  main
    java
      com.example.demo
        DemoApplication
    resources
      application.properties
  test
    java
      com.example.demo
        DemoApplicationTests
```

Figure 2-1. Maven project structure within a Spring Boot application

---

NOTE

For more details about Maven's expected project structure, refer to The Maven Project's Introduction to the Standard Directory Layout.

If there comes a time when Maven's project conventions and/or tightly structured approach to builds become too constrictive, there is another excellent option.

Gradle

Gradle is another popular option for building Java Virtual Machine (JVM) projects. First released in 2008, Gradle leverages a Domain Specific Language (DSL) to produce a *build.gradle* build file that is both minimal and flexible. An example of a Gradle build file for a Spring Boot application follows.

```
plugins {
        id 'org.springframework.boot' version '2.4.0'
        id 'io.spring.dependency-management' version '1.0.10.RELEASE'
        id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'
```

```
repositories {
        mavenCentral()
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter'
        testImplementation 'org.springframework.boot:spring-boot-starter-tes
}

test {
        useJUnitPlatform()
}
```

Gradle allows you, the developer, to choose to use either the Groovy or the Kotlin programming languages for a DSL. It also offers several features meant to reduce your time waiting for a project to build, such as the following:

- Incremental compilation of Java classes
- Compile avoidance for Java (in cases where no changes occurred)
- A dedicated daemon for project compilation

Choosing Between Maven and Gradle

Your choice of build tool may not sound like much of a choice at this point. Why not simply choose Gradle?

Maven's more rigid declarative (some might say opinionated) approach keeps things incredibly consistent from project to project, environment to environment. If you follow the Maven way, few issues typically crop up, leaving you to focus on your code with little fussing with the build.

As a build system built around programming/scripting, Gradle also occasionally has issues digesting initial releases of new language versions. The Gradle team is responsive and typically dispatches these issues with great haste, but if you prefer to (or must) dive immediately into early access language releases, this warrants consideration.

Gradle can be faster for builds—and sometimes *significantly* faster, especially in larger projects. That said, for your typical microservices-based project, build times aren't likely to differ by that much between similar Maven and Gradle projects.

Gradle's flexibility can be a breath of fresh air for simple projects and projects with very complex build requirements. But especially in those complex projects, Gradle's additional flexibility can result in more time spent tweaking and troubleshooting when things aren't working the way you might expect. TANSTAAFL (There Ain’t No Such Thing as a Free Lunch).

Spring Boot supports both Maven and Gradle, and if you use the Initializr (to be covered in a section that follows), the project and desired build file are created for you, to get you up and running quickly. In short, try both, then choose what works best for you. Spring Boot will happily support you either way.

Java or Kotlin?

While there are many languages available for use on the JVM, two enjoy the most widespread use. One is the original JVM language, Java; the other is a relative newcomer to the space, Kotlin. Both are full first-class citizens in Spring Boot.

Java

Depending on whether you consider the public 1.0 release or the project origin as its official birthdate, Java has been around for 25 or 30 years, respectively. It’s anything but stagnant, though. Since September 2017, Java has been on a six-month release cycle, resulting in more frequent feature improvements than before. The maintainers have cleaned up the codebase and have pruned features obviated by new ones, as well as introduced vital features driven by the Java community. Java is more vibrant than ever.

That lively pace of innovation, combined with Java’s longevity and consistent focus on backward compatibility, means that there are countless Java shops maintaining and creating critical Java applications daily around the world. Many of those applications use Spring.

Java forms the bedrock-solid foundation of nearly the entire Spring codebase, and as such, it’s a great choice to use for building your Spring Boot applications. Examining the code for Spring, Spring Boot, and all related projects is a simple matter of visiting GitHub where it’s hosted and viewing it there or cloning the project to review offline. And with the availability of an abundance of example code, sample projects, and “Getting Started” guides written using Java, writing Spring Boot apps using Java may be better supported than any other toolchain combination on the market.

Kotlin

Relatively speaking, Kotlin is the new kid on the block. Created by JetBrains in 2010 and made public in 2011, Kotlin was created to address perceived gaps in Java's usability. Kotlin was designed from the beginning to be:

Concise

> Kotlin requires minimal code to clearly communicate intent to the compiler (and oneself and other developers).

Safe

> Kotlin eliminates null-related errors by eliminating the possibility of null values *by default*, unless the developer specifically overrides behavior to allow for them.

Interoperable

> Kotlin aims for frictionless interoperability with all existing JVM, Android, and browser libraries.

Tool-friendly

> Build Kotlin applications in numerous Integrated Development Environments (IDEs) or from the command line, just like Java.

Kotlin's maintainers extend the language's capabilities with great care but also with great velocity. Without 25+ years of language compatibility as a core design focus, they've moved quickly to add very useful capabilities that are likely to appear in Java some versions later.

In addition to being concise, Kotlin is also a very fluent language. Without diving into too many details yet, several language features contribute to this linguistic elegance, among them `extension functions` and `infix notation`. I'll discuss this idea in more depth later, but Kotlin makes syntax options like this possible:

```kotlin
infix fun Int.multiplyBy(x: Int): Int { ... }

// calling the function using the infix notation
1 multiplyBy 2

// is the same as
1.multiplyBy(2)
```

As you might imagine, the ability to define your own, more fluent "language within a language" can be a boon to API design. Combined with

Kotlin's concision, this can make Spring Boot applications written in Kotlin even shorter and more readable than their Java counterparts, with no loss in communication of intent.

Kotlin has been a full first-class citizen in Spring Framework since version 5.0 was released in autumn of 2017, with full support propagating through Spring Boot (spring 2018) and other component projects ever since. Additionally, all Spring documentation is being expanded to include examples in both Java *and* Kotlin. This means that effectively, you can write entire Spring Boot applications with Kotlin as easily as with Java.

Choosing Between Java and Kotlin

The amazing thing is that you don't actually have to choose. Kotlin compiles to the same bytecode output that Java does; and since Spring projects can be created that include both Java source files and Kotlin, and can call both compilers with ease, you can use whichever makes more sense to you even *within the same project*. How's that for having your cake and eating it too?

Of course, if you prefer one over the other or have other personal or professional strictures, you're obviously able to develop entire applications in one or the other. It's good to have choices, no?

Choosing a Version of Spring Boot

For production applications, you should always use the current version of Spring Boot with the following temporary and narrow exceptions:

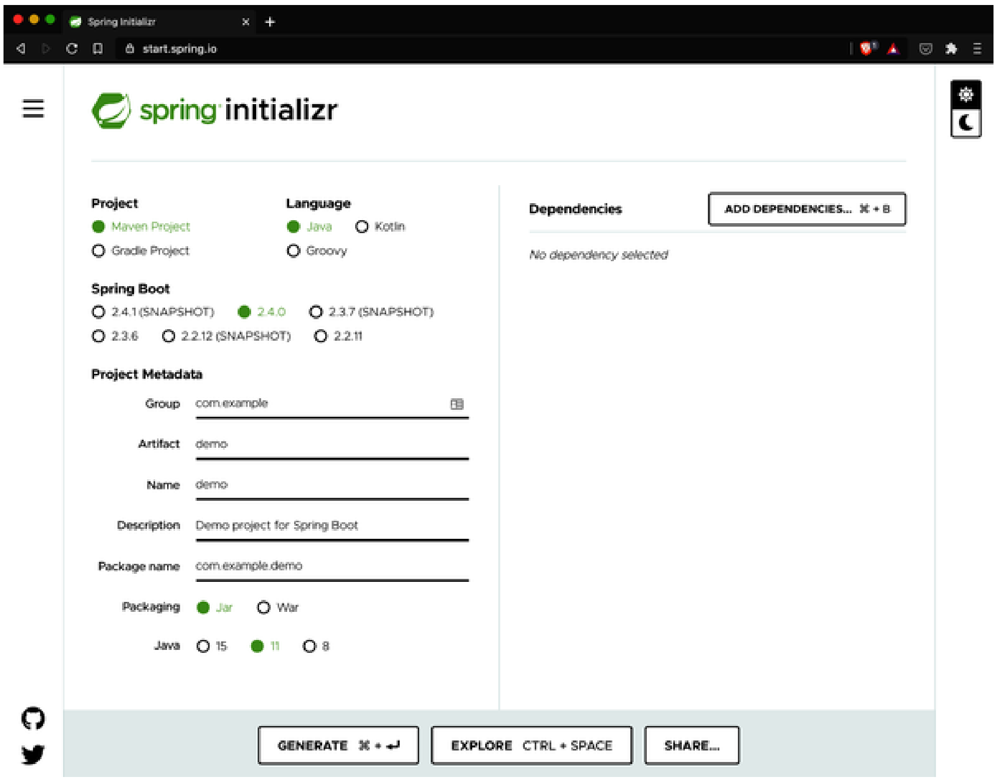
- You're currently running an older version but are upgrading, retesting, and deploying your applications in some order such that you simply haven't reached this particular app yet.
- You're currently running an older version, but there is an identified conflict or bug you've reported to the Spring team and are instructed to wait for an update to Boot or a dependency in question.
- You need to utilize features in a snapshot, milestone, or release candidate pre-GA (General Availability) version and are willing to accept the risks inherent with code that hasn't yet been declared GA, i.e., "ready for production use."

NOTE

Snapshot, milestone, and Release Candidate (RC) versions are extensively tested prior to publication, so a great deal of rigor has already gone into ensuring their stability. Until the full GA version is approved and published, though, there is always the potential for API changes, fixes, etc. The risks to your application are low, but you'll have to decide for yourself (and test and confirm) if those risks are manageable when you consider using *any* early-access software.

The Spring Initializr

There are many ways to create a Spring Boot application, but most lead back to a single starting point: the Spring Initializr, shown in Figure 2-2.



Project: [✓ Maven Project] Gradle Project

Language: [✓ Java] Kotlin Groovy

Spring Boot: 2.4.1 (SNAPSHOT) [✓ 2.4.0] 2.3.7 (SNAPSHOT) 2.3.6 2.2.12 (SNAPSHOT) 2.2.11

Project Metadata

Group	com.example
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
Package name	com.example.demo
Packaging	[✓ Jar] War
Java	15 [✓ 11] 8

Dependencies

No dependency selected

Figure 2-2. The Spring Initializr

Sometimes referred to simply by its URL, `start.spring.io`, the Spring Initializr can be accessed from project creation wizards within prominent IDEs, via the command line, or most often via a web browser. Using the web browser provides a few extra useful features that can't (yet) be accessed via other avenues.

To begin creating a Spring Boot project in the "Best Possible Way," point your browser to *https://start.spring.io*. From there, we'll choose a few options and get started.

INSTALLING JAVA

I assume that if you've reached this point you've already installed a current version of the Java Development Kit (JDK)—sometimes referred to as *Java Platform, Standard Edition*—on your machine. If you haven't yet installed Java, you'll need to do so before proceeding.

Providing detailed instructions for how to do so lies outside the scope of this book, but a few recommendations wouldn't hurt either, right? :)

I've found that the easiest way to install and manage one or more JDKs on your machine is by using SDKMAN!. This package manager also facilitates the installation of the Spring Boot Command Line Interface (CLI) you'll use later (and many, many other tools), so it's an incredibly useful utility app to have. Following the instructions at *https://sdkman.io/install* will get you ready to roll.

WARNING

SDKMAN! is written in bash (the Unix/Linux shell) script, and as such, it installs and works natively on MacOS and Linux as well as other operating systems with Unix or Linux foundations. SDKMAN! will run on Windows as well but not natively; in order to install and run SDKMAN! in a Windows environment, you must first install the Windows Subsystem for Linux (WSL), Git Bash for Windows plus MinGW. Please see the SDKMAN! install page linked previously for details.

From SDKMAN!, it's a matter of installing the desired version of Java using `sdk list java` to view options, then `sdk install java <insert_desired_java_here>` to install. Numerous great options exist, but to start, I'd recommend you choose the current Long Term Support (LTS) version packaged by AdoptOpenJDK with the Hotspot JVM, e.g., `11.0.7.hs-adpt`.

If you prefer not to use SDKMAN! for whatever reason, you can also simply choose to download and install a JDK directly from *https://adoptopenjdk.net*. Doing so will get you up and running, but it makes updates more difficult and doesn't help you in the future with updating or if you need to manage multiple JDKs.

To get started with the Initializr, we first choose the build system we plan to use with our project. As mentioned previously, we have two great options: Maven and Gradle. Let's choose Maven for this example.

Next, we'll choose Java as the (language) basis for this project.

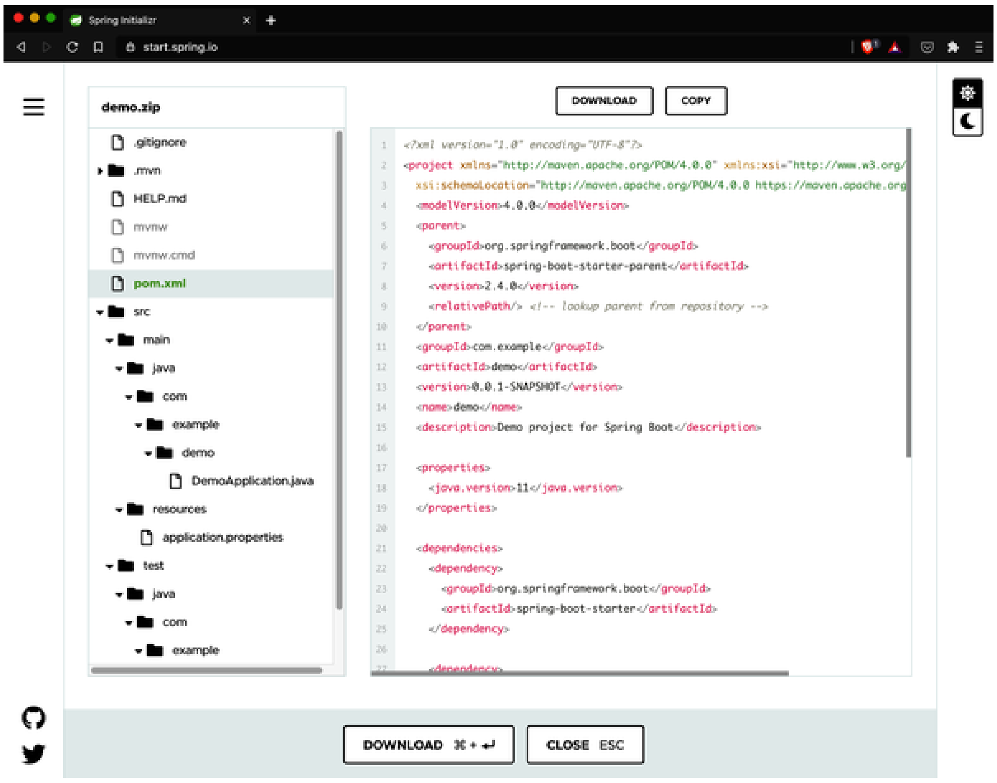
As you may have noticed already, the Spring Initializr selects enough defaults for the options presented to create a project with no input from you whatsoever. When you reached this web page, Maven and Java were both already preselected. The current version of Spring Boot is as well, and for this—and most—projects, that is what you'll want to select.

We can leave the options under Project Metadata as they are without issue, although we'll modify them for future projects.

And for now, we also don't include any dependencies. This way, we can focus on the mechanics of project creation, not any particular outcome.

Before generating that project, though, there are a couple of really nice features of the Spring Initializr I'd like to point out, along with one sidenote.

If you'd like to examine your project's metadata and dependency details prior to project generation based on your current selections, you can click the Explore button or use the keyboard shortcut, Ctrl+Space, to open Spring Initializr's Project Explorer (shown in Figure 2-3). The Initializr will then present you with the project structure and build file that will be included in the compressed (*.zip*) project you're about to download. You can review the directory/package structure, application properties file (more on this later), and the project properties and dependencies specified in your build file: since we're using Maven for this project, ours is *pom.xml*.



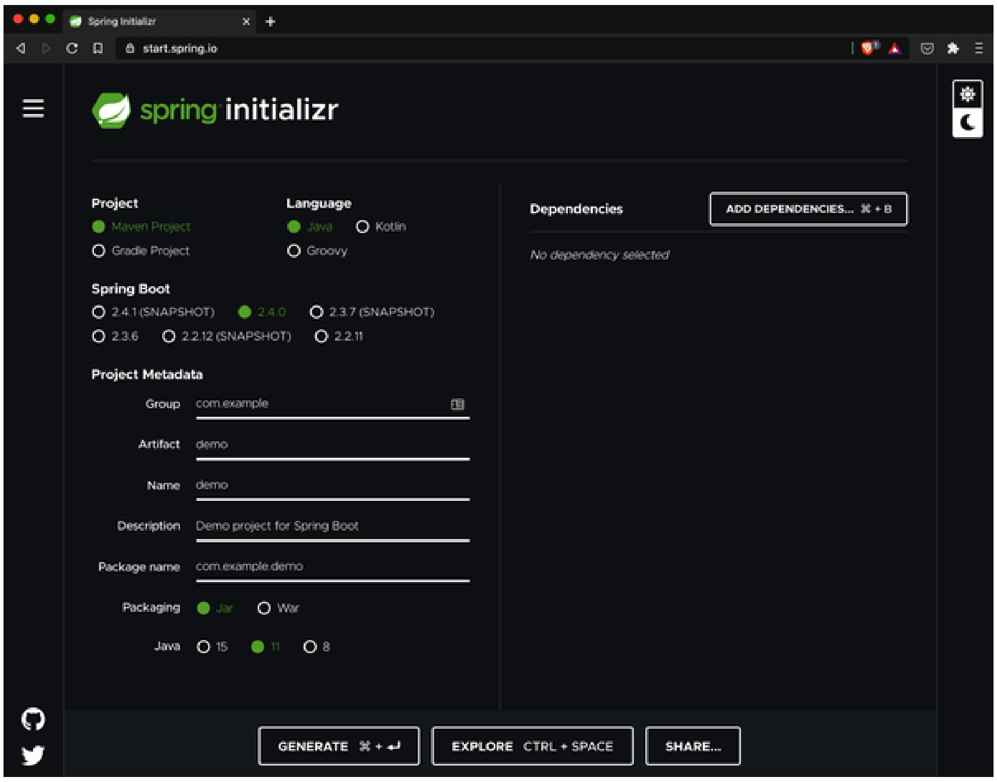
```
demo.zip
  .gitignore
▸ .mvn
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
▾ src
  ▾ main
    ▾ java
      ▾ com
        ▾ example
          ▾ demo
              DemoApplication.java
    ▾ resources
        application.properties
  ▾ test
    ▾ java
      ▾ com
        ▾ example
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>11</java.version>
  </properties>

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

    <dependency>
```

Figure 2-3. Spring Initializr's Project Explorer

This is a quick and handy way to verify project configuration and dependencies before downloading, extracting, and loading into your IDE your brand-new, empty project.

Another smaller feature of the Spring Initializr, but one that has been welcomed by numerous developers, is dark mode. By clicking on the `Dark UI` toggle at the top of the page as shown in Figure 2-4, you switch to Initializr's dark mode and make that the default each time you visit the page. It's a small feature, but if you keep your machine in dark mode everywhere else, it certainly makes loading the Initializr less jarring and more pleasant. You'll want to keep coming back!



Project: [✓ Maven Project] Gradle Project

Language: [✓ Java] Kotlin Groovy

Spring Boot: 2.4.1 (SNAPSHOT) [✓ 2.4.0] 2.3.7 (SNAPSHOT) 2.3.6 2.2.12 (SNAPSHOT) 2.2.11

Project Metadata

Group: com.example

Artifact: demo

Name: demo

Description: Demo project for Spring Boot

Package name: com.example.demo

Packaging: [✓ Jar] War

Java: 15 [✓ 11] 8

Dependencies: *No dependency selected*

Figure 2-4. The Spring Initializr, in dark mode!

NOTE

Other than the main application class and its main method, plus an empty test, the Spring Initializr doesn't generate code for you; it generates the *project* for you, per your guidance. It's a small distinction but a very important one: code generation has wildly varied results and often hamstrings you the moment you begin making changes. By generating the project structure, including the build file with specified dependencies, the Initializr provides you a running start to write the code you need to leverage Spring Boot's autoconfiguration. Autoconfig gives you superpowers without the straitjacket.

Next, click the Generate button to generate, package, and download your project, saving it to your chosen location on your local machine. Then navigate to that downloaded *.zip* file and unzip it to prepare to develop your application.

Straight Outta Commandline

If you happily spend as much time as possible on the command line or wish to eventually script project creation, the Spring Boot Command Line Interface (CLI) was made for you. The Spring Boot CLI has many powerful capabilities, but for now, we’ll limit our focus to creating a new Boot project.

INSTALLING THE SPRING BOOT CLI

Perhaps the easiest way to install the Spring Boot CLI is via SDKMAN!, as with your JDK, Kotlin utilities, and more. From your terminal window, you can run

```
sdk list
```

to see all of the various packages available for installation; Figure 2-5 shows the Spring Boot CLI entry. Next, run

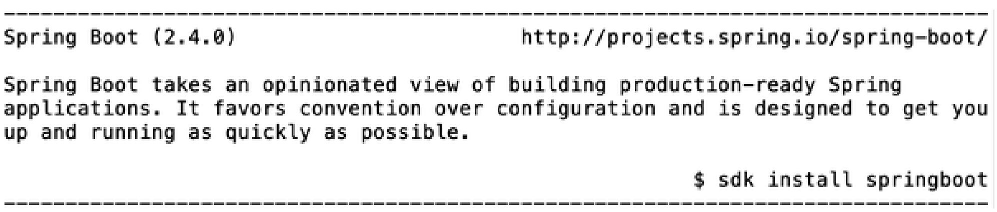
```
sdk list springboot
```

to see available versions of the Spring Boot CLI, then install the most recent (current) one with

```
sdk install springboot
```

If you don't provide a specific version identifier with the SDKMAN! command `sdk install <tool> <version_identifier>`, SDKMAN! typically installs the latest recommended production version of the language/tool. This has different meanings for different supported packages; as an example, the latest Long Term Support (LTS) version of Java will be installed, not a more recent (non-LTS) version that may be available. This is because a new numbered version of Java is released every six months, and periodically, a version is designated an LTS release—meaning that there are often one or more newer releases that are officially supported for only six months each (for feature evaluation, testing, or even production deployments), while a particular LTS release is fully supported with updates and bug fixes.

This note is a bit of a generalization that can vary somewhat between JDK providers, although most don't stray far (if at all) from the customary designations. Entire talks have been devoted to the details, but for our purposes here, they have no effect whatsoever.



```
--------------------------------------------------------------------------------
Spring Boot (2.4.0)                          http://projects.spring.io/spring-boot/

Spring Boot takes an opinionated view of building production-ready Spring
applications. It favors convention over configuration and is designed to get you
up and running as quickly as possible.

                                                          $ sdk install springboot
--------------------------------------------------------------------------------
```

Figure 2-5. The Spring Boot CLI on SDKMAN!

Once you've installed the Spring Boot CLI, you can create the same project we just created with the following command:

```
spring init
```

To extract the zipped project into a directory named *demo*, you can execute the following command:

```
unzip demo.zip -d demo
```

Wait, how can it be so easy? In one word, defaults. The Spring CLI uses the same default settings as the Spring Initializr (Maven, Java, etc.), allowing you to provide arguments only for values you wish to change. Let's specifically provide values for a few of those defaults (and add a helpful twist for project extraction) just to better see what is involved:

```
spring init -a demo -l java --build maven demo
```

We're still initializing a project using the Spring CLI, but now we're providing the following arguments:

- `-a demo` (or `--artifactId demo`) allows us to provide an artifact ID for the project; in this case, we call it "demo."
- `-l java` (or `--language java`) lets us specify Java, Kotlin, or Groovy[1] as the primary language we'll use for this project.
- `--build` is the flag for the build system argument; valid values are `maven` and `gradle`.
- `-x demo` requests the CLI to extract the resultant project *.zip* file returned by the Initializr; note that the `-x` is optional, and specifying a text label without an extension (as we've done here) is helpfully inferred to be an extraction directory.

NOTE

All of these options can be reviewed further by executing `spring help init` from the command line.

Things get a bit more involved when specifying dependencies. As you might imagine, it's hard to beat the ease of choosing them from the "menu" presented by the Spring Initializr. But the flexibility of the Spring CLI is handy in the extreme for quick starts, scripting, and build pipelines.

One more thing: by default, the CLI leverages the Initializr to provide its project-building capabilities, which means projects created via either mechanism (CLI or via the Initializr web page) are identical. That consistency is absolutely essential in a shop that directly uses the Spring Initializr's capabilities.

Occasionally, though, an organization tightly controls what dependencies their developers can even use to create projects. To be completely honest, this approach saddens me and feels very timebound, impeding an organization's agility and user/market responsiveness. If you're in such an organization, this complicates your ability to "get the job done" on anything you set out to accomplish.

That being the case, it is possible to create your own project generator (even cloning the repository for the Spring Initializr) and use that directly via the resultant web page…or only expose the REST API portion and utilize that from the Spring CLI. To do so, just add this parameter to the commands shown earlier (replacing with your valid URL, of course):

```
--target https://insert.your.url.here.org
```

Staying In Integrated Development Environments (IDEs)

However you create your Spring Boot project, you'll need to open it and write some code to create a useful application.

There are three major IDEs and numerous text editors that do a respectable job of supporting you as a developer. IDEs include, but are not limited to, Apache NetBeans, Eclipse, and IntelliJ IDEA. All three are open source software (OSS), and all three are free in many circumstances.[2]

In this book, as in my daily life, I primarily use IntelliJ Ultimate Edition. There isn't really a wrong choice as much as a personal preference (or organizational mandate or preference) in choosing an IDE, so please use what fits you and your tastes best. Most concepts transfer pretty well among the major offerings.

There are also several editors that have garnered a large following among devs. Some, like Sublime Text, are paid applications that have fierce followings due to their quality and longevity. Other more recent entrants to the field, like Atom (created by GitHub, which is now owned by Microsoft) and Visual Studio Code (shortened to VSCode, created by Microsoft) are gaining capabilities and loyal followings rapidly.

In this book, I occasionally use VSCode or its counterpart built from the same codebase but with telemetry/tracking disabled, VSCodium. In order to support some of the features most developers expect and/or require from their development environment, I add the following extensions to VSCode/VSCodium:

Spring Boot Extension Pack (Pivotal)

> This includes several other extensions, including `Spring Initializr Java Support`, `Spring Boot Tools`, and `Spring Boot Dashboard`, which facilitate the creation, editing, and management of Spring Boot applications within VSCode, respectively.

Debugger for Java (Microsoft)

> Dependency of the Spring Boot Dashboard.

IntelliJ IDEA Keybindings (Keisuke Kato)

> Because I primarily use IntelliJ, this makes it easier for me to switch between the two more easily.

Language Support for Java™ (Red Hat)

> Dependency of Spring Boot Tools.

Maven for Java (Microsoft)

> Facilitates the use of Maven-based projects.

There are other extensions you may find useful for wrangling XML, Docker, or other ancillary technologies, but these are the essentials for our current purposes.

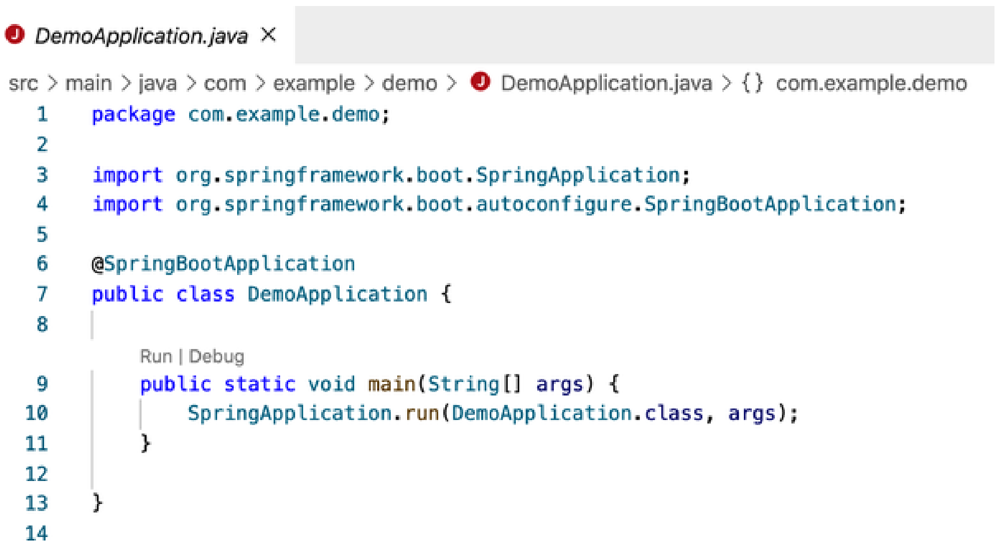
Continuing with our Spring Boot project, you'll next want to open it in your chosen IDE or text editor. For most of the examples in this book, we'll use IntelliJ IDEA, a very capable IDE (written in Java and Kotlin) produced by JetBrains. If you've already associated your IDE with project build files, you can double-click on the `pom.xml` file in your project's directory (using Finder on the Mac, File Explorer on Windows, or one of the various File Managers on Linux) and load the project into your IDE automatically. If not, you can open the project from within your IDE or editor in the manner recommended by its developers.

NOTE

Many IDEs and editors offer a way to create command line shortcuts to launch and load, bringing up your project with one short command. Examples include IntelliJ's `idea`, VSCode's/VSCodium's `code`, and Atom's `atom` shortcuts.

Cruising Down main()

Now that we've loaded the project in our IDE (or editor), let's take a look at what makes a Spring Boot project (Figure 2-6) just a bit different from a standard Java application.


DemoApplication.java ×
src > main > java > com > example > demo > DemoApplication.java > {} com.example.demo
Run | Debug

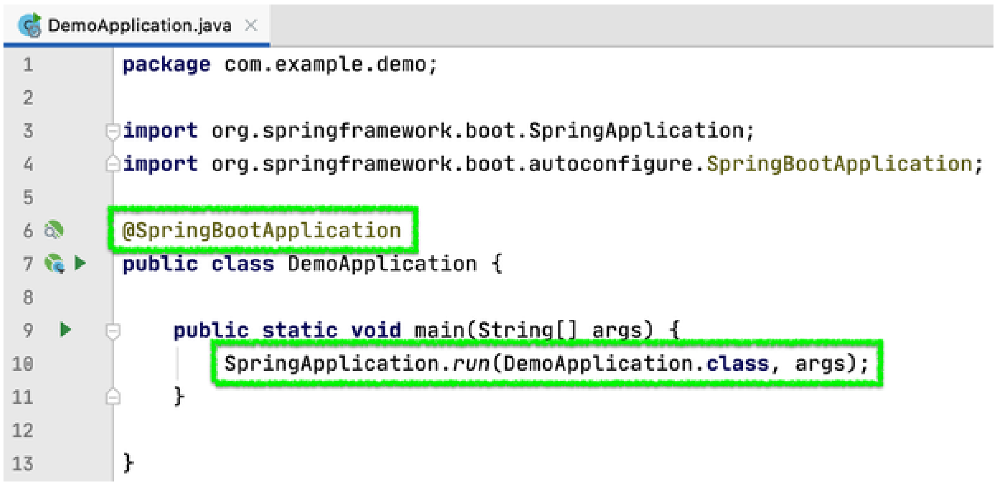
```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

Figure 2-6. Our Spring Boot demo application's main application class

A standard Java application contains (by default) an empty `public static void main` method. When we execute a Java application, the JVM searches for this method as the app's starting point, and without it, application startup fails with an error like this one:

```
Error:
Main method not found in class PlainJavaApp, please define the main method a
        public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Of course, you can place code to be executed upon application startup in a Java class's main method, and a Spring Boot application does exactly that. Upon startup, a Spring Boot app checks the environment, configures the application, creates the initial context, and launches the Spring Boot application. It does this via a single top-level annotation and a single line of code, as shown in Figure 2-7.



DemoApplication.java

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

Figure 2-7. The essence of a Spring Boot application

We'll dive under the covers of these mechanisms as the book unfolds. For now, suffice it to say that Boot takes a lot of tedious application setup off your hands during application startup *by design* and *by default* so that you can quickly get down to the business of writing meaningful code.

Summary

This chapter has examined some of the first-class choices you have in creating Spring Boot applications. Whether you prefer to build your projects using Maven or Gradle, write code in Java or Kotlin, or create projects from the web interface provided by the Spring Initializr or its command line partner, the Spring Boot CLI, you have the full power and ease of Spring Boot at your fingertips without compromise. You can also work with Boot projects using an impressive variety of IDEs and text editors with top-notch Spring Boot support.

As covered here and in Chapter 1, the Spring Initializr works hard for you in getting your project created quickly and easily. Spring Boot contributes meaningfully throughout the development life cycle with the following features:

- Simplified dependency management, which comes into play from project creation through development and maintenance
- Autoconfiguration that dramatically reduces/eliminates the boilerplate you might otherwise write before working on the problem domain
- Simplified deployment that makes packaging and deployment a breeze

And all of these capabilities are fully supported regardless of build system, language, or toolchain choices you make along the way. It's an amazingly flexible and powerful combination.

In the next chapter, we'll create our first really meaningful Spring Boot application: an app that provides a REST API.

[1] Groovy support is still provided within Spring Boot but is nowhere near as widely used as Java or Kotlin.

[2] There are two options available: Community Edition (CE) and Ultimate Edition (UE). Community Edition supports Java and Kotlin app development, but to have access to all available Spring support, you must use Ultimate Edition. Certain use cases qualify for a free license for UE, or you can of course also purchase one. Additionally, all three provide excellent support for Spring Boot applications.