Simple Spring Boot Application

jim stafford

Fall 2022 v2022-08-31: Built: 2022-12-07 06:11 EST

Table of Contents

Chapter 1. Introduction

This material makes the transition from a creating and executing a simple Java main application to a Spring Boot application.

1.1. Goals

The student will learn:

- foundational build concepts for simple, Spring Boot Application

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. extend the standard Maven `jar` module packaging type to include core Spring Boot dependencies
2. construct a basic Spring Boot application
3. build and execute an executable Spring Boot JAR
4. define a simple Spring component and inject that into the Spring Boot application

Chapter 2. Spring Boot Maven Dependencies

Spring Boot provides a `spring-boot-starter-parent` (gradle source, pom.xml) pom that can be used as a parent pom for our Spring Boot modules. [1] This defines version information for dependencies and plugins for building Spring Boot artifacts — along with an opinionated view of how the module should be built.

`spring-boot-starter-parent` inherits from a `spring-boot-dependencies` (gradle source, pom.xml) pom that provides a definition of artifact versions without an opinionated view of how the module is built. This pom can be imported by modules that already inherit from a local Maven parent — which would be common. This is the demonstrated approach we will take here. We will also include demonstration of how the build constructs are commonly spread across parent and local poms.

Spring Boot has converted over to gradle and posts a pom version of the gradle artifact to Maven central repository as a part of their build process.

[1] Spring Boot and Build Systems, Pivotal

Chapter 3. Parent POM

We are likely to create multiple Spring Boot modules and would be well-advised to begin by creating a local parent pom construct to house the common passive definitions. By passive definitions (versus active declarations), I mean definitions for the child poms to use if needed versus mandated declarations for each child module. For example, a parent pom may define the JDBC driver to use when needed but not all child modules will need a JDBC driver nor a database for that matter. In that case, we do not want the parent pom to actively declare a dependency. We just want the parent to passively define the dependency that the child can optionally choose to actively declare. This construct promotes consistency among all of the modules.

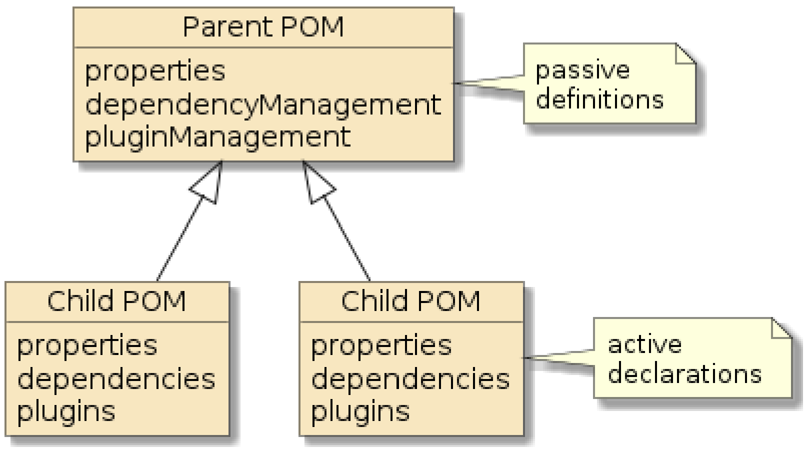


Figure 1. Parent/Child Pom Relationship and Responsibilities

"Root"/parent poms should define dependencies and plugins for consistent re-use among child poms and use dependencyManagement and pluginManagement elements to do so.

"Child"/concrete/leaf poms declare dependencies and plugins to be used when building that module and try to keep dependencies to a minimum.

"Prototype" poms are a blend of root and child pom concepts. They are a nearly-concrete, parent pom that can be extended by child poms but actively declare a select set of dependencies and plugins to allow child poms to be as terse as possible.

3.1. Define Version for Spring Boot artifacts

Define the version for Spring Boot artifacts to use. I am using a technique below of defining the value in a property so that it is easy to locate and change as well as re-use elsewhere if necessary.

Explicit Property Definition

```
# Place this declaration in an inherited parent pom
<properties>
    <springboot.version>2.7.0</springboot.version> ①
```

```
</properties>
```

① default value has been declared in imported `ejava-build-bom`

Property values can be overruled at build time by supplying a system property on the command line "-D(name)=(value)"

3.2. Import springboot-dependencies-plugin

Import `springboot-dependencies-plugin`. This will define dependencyManagement for us for many artifacts that are relevant to our Spring Boot development.

```
# Place this declaration in an inherited parent pom
<dependencyManagement> ①
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${springboot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

① import is within `examples-root` for class examples, which is a grandparent of this example

Chapter 4. Local Child/Leaf Module POM

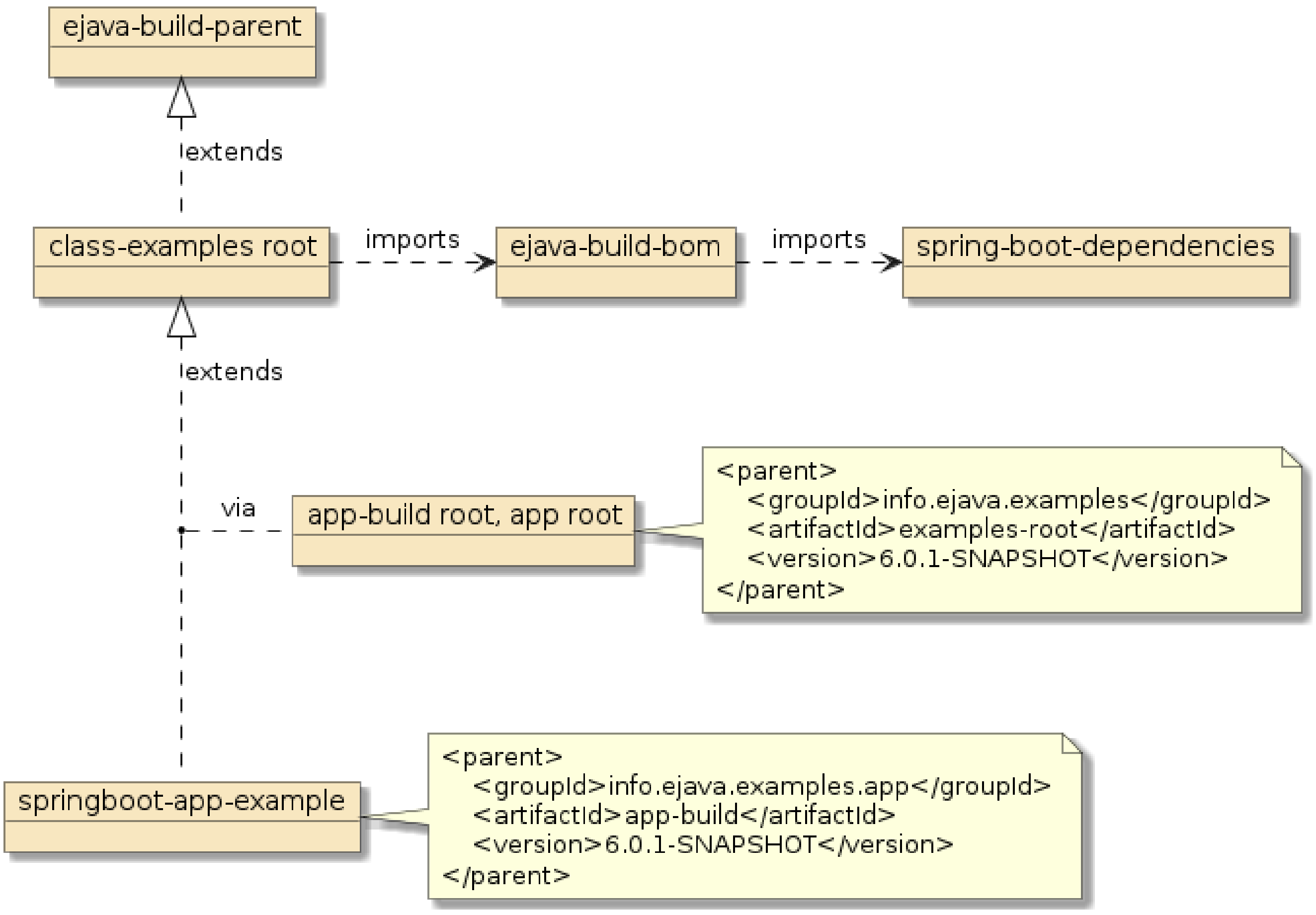
The local child module pom.xml is where the module is physically built. Although Maven modules can have multiple levels of inheritance — where each level is a child of their parent, the child module I am referring to here is the leaf module where the artifacts are meant to be really built. Everything defined above it is primarily used as a common definition (thru dependencyManagement and pluginManagement) to simplify the child pom.xml and to promote consistency among sibling modules. It is the job of the leaf module to activate these definitions that are appropriate for the type of module being built.

4.1. Declare pom inheritance in the child pom.xml

Declare pom inheritance in the child pom.xml to pull in defintions from parent pom.xml.

```
# Place this declaration in the child/leaf pom building the JAR archive
<parent>
    <groupId>(parent groupId)</groupId>
    <artifactId>(parent artifactId)</artifactId>
    <version>(parent version)</version>
</parent>
```

The following diagram shows the parent/child relationship between the `springboot-app-example` and the `class-example-root` pom and the parent's relationships.



```
<parent>
   <groupId>info.ejava.examples</groupId>
   <artifactId>examples-root</artifactId>
   <version>6.0.1-SNAPSHOT</version>
</parent>
```

```
<parent>
   <groupId>info.ejava.examples.app</groupId>
   <artifactId>app-build</artifactId>
   <version>6.0.1-SNAPSHOT</version>
</parent>
```

Figure 2. SpringBoot App Example POM Tree

4.2. Declare dependency on artifacts used

Realize the parent definition of the `spring-boot-starter` dependency by declaring it within the child dependencies section. For where we are in this introduction, only the above dependency will be necessary. The imported `spring-boot-dependencies` will take care of declaring the version#

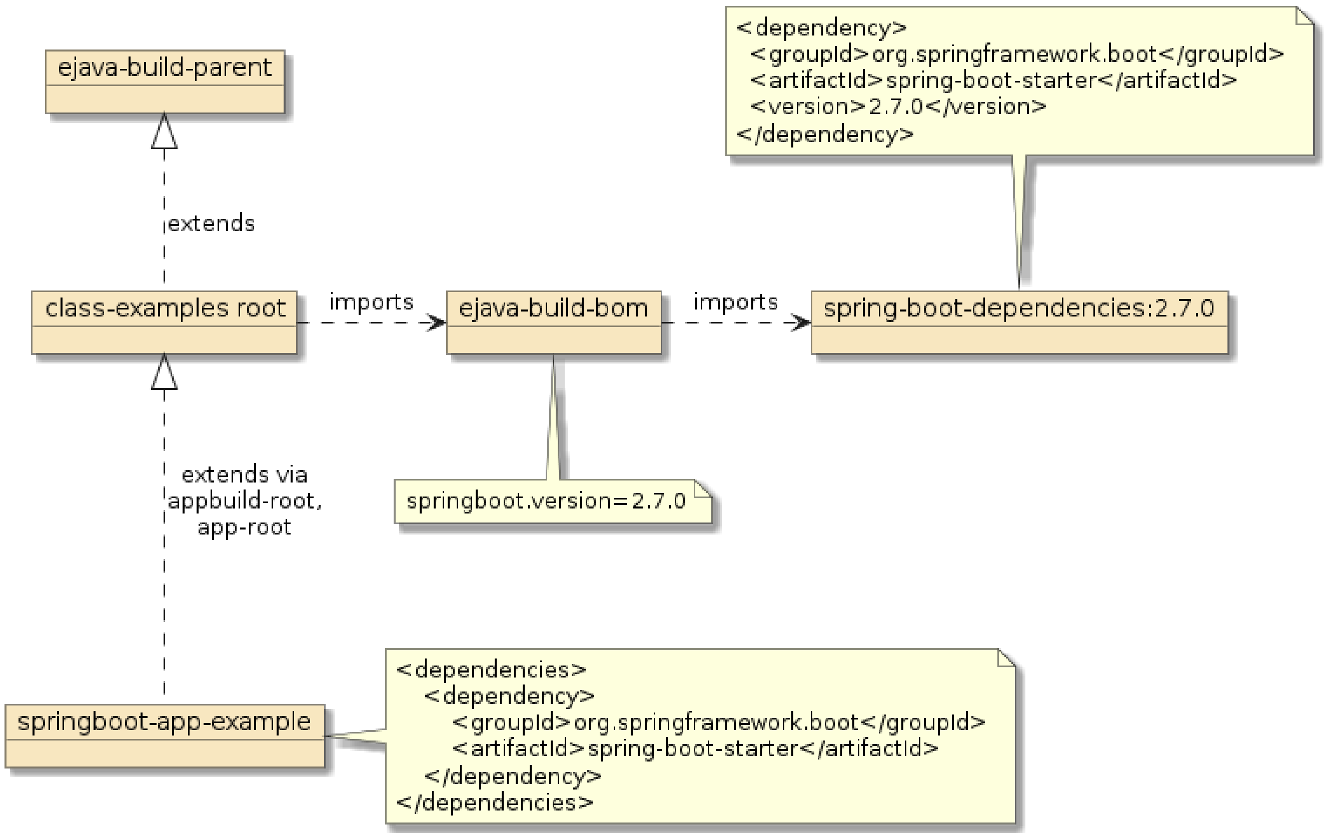
```
# Place this declaration in the child/leaf pom building the JAR archive
<dependencies>
  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
      <!--version --> ①
  </dependency>
</dependencies>
```

① parent has defined (using `import` in this case) the version for all children to consistently use

The figure below shows the parent poms being the source of the passive dependency definitions and the child being the source of the active dependency declarations.

- the parent is responsible for defining the version# for dependencies used
- the child is responsible for declaring what dependencies are needed and adopts the parent version definition

An upgrade to a future dependency version should not require a change of a child module declaration if this pattern is followed.



```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <version>2.7.0</version>
</dependency>
```

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

Figure 3. Class Examples dependencyManagement

Chapter 5. Simple Spring Boot Applicaton Java Class

With the necessary dependencies added to our build classpath, we now have enough to begin defining a simple Spring Boot Application.

```
package info.ejava.springboot.examples.app.build.springboot;

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

@SpringBootApplication ③
public class SpringBootApp {
    public static final void main(String...args) { ①
        System.out.println("Running SpringApplication");

        SpringApplication.run(SpringBootApp.class, args); ②

        System.out.println("Done SpringApplication");
    }
}
```

① Define a class with a static main() method

② Initiate Spring applicaton bootstrap by invoking `SpringApplication.run()` and passing a) application class and b) args passed into main()

③ Annotate the class with `@SpringBootApplication`

Startup can, of course be customized (e.g., change the printed banner, registering event listeners)

5.1. Module Source Tree

The source tree will look similar to our previous Java main example.

```
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- info
    |   |       `-- ejava
    |   |           `-- examples
    |   |               `-- app
    |   |                   `-- build
    |   |                       `-- springboot
    |   |                           `-- SpringBootApp.java
    |   `-- resources
    `-- test
        |-- java
        `-- resources
```

5.2. @SpringBootApplication Aggregate Annotation

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

@SpringBootApplication
public class SpringBootApp {
}
```

The @SpringBootApplication annotation is a compound class-level annotation aggregating the following annotations.

- @ComponentScan - legacy Spring annotation that configures component scanning to include or exclude looking thru various packages for classes with component annotations
    - By default, scanning will start with the package declaring the annotation and work its way down from there
- @SpringBootConfiguration - like legacy Spring @Configuration annotation, it signifies the class can provide configuration information. Unlike `@Configuration`, there can be only one `@SpringBootConfiguration` per application and it is normally supplied by `@SpringBootApplication` except in some integration tests.
    - Classes annotated with `@Configuration` contain factory `@Bean` definitions.
- @EnableAutoConfiguration - Allows Spring to perform auto-configuration based on the classpath, beans defined by the application, and property settings.

The class annotated with @SpringBootApplication is commonly located in a Java package that is above all other Java packages containing components for the application.

Chapter 6. Spring Boot Executable JAR

At this point we can likely execute the Spring Boot Application within the IDE but instead, lets go back to the pom and construct a JAR file to be able to execute the application from the command line.

6.1. Building the Spring Boot Executable JAR

We saw earlier how we could build a standard executable JAR using the `maven-jar-plugin`. However, there were some limitations to that approach — especially the fact that a standard Java JAR cannot house dependencies to form a self-contained classpath and Spring Boot will need additional JARs to complete the application bootstrap. Spring Boot uses a custom executable JAR format that can be built with the aid of the spring-boot-maven-plugin. Lets extend our pom.xml file to enhance the standard JAR to be a Spring Boot executable JAR.

6.1.1. Declare spring-boot-maven-plugin

The following snippet shows the configuration for a `spring-boot-maven-plugin` that defines a default execution to build the Spring Boot executable JAR for all child modules that declare using it. In addition to building the Spring Boot executable JAR, we are setting up a standard in the parent for all children to have their follow-on JAR classified separately as a `bootexec`. `classifier` is a core Maven construct and is meant to lable sibling artifacts to the original Java JAR for the module. Other types of `classifiers` are `source`, `schema`, `javadoc`, etc. `bootexec` is a value we made up.

By default, the `repackage` goal would have replaced the Java JAR with the Spring Boot executable JAR. That would have left an ambiguous JAR artifact in the repository — we would not easily know its JAR type. This will help eliminate dependency errors during the semester when we layer `N+1` assignments on top of layer `N`. Only standard Java JARs can be used in classpath dependencies.

spring-boot-maven-plugin with classifier

```
<properties>
    <spring-boot.classifier>bootexec</spring-boot.classifier>
</properties>
...
<build>
    <plugins>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
          <configuration>
              <classifier>${spring-boot.classifier}</classifier> ④
          </configuration>
          <executions>
              <execution>
                  <id>build-app</id> ①
                  <phase>package</phase> ②
                  <goals>
                    <goal>repackage</goal> ③
                  </goals>
              </execution>
          </executions>
        </plugin>
        ...
    </plugins>
</build>
```

① `id` used to describe execution and required when having more than one

② `phase` identifies the maven goal in which this plugin runs

③ `repackage` identifies the goal to execute within the `spring-boot-maven-plugin`

④ adds a -`bootexec` to the executable JAR's name

We can do much more with the `spring-boot-maven-plugin` on a per-module basis (e.g., run the application from within Maven). We are just starting at construction at this point.

6.1.2. Build the JAR

```
$ mvn clean package

[INFO] Scanning for projects...
...
[INFO] --- maven-jar-plugin:3.2.2:jar (default-jar) @ springboot-app-example ---
[INFO] Building jar: .../target/springboot-app-example-6.0.1-SNAPSHOT.jar ①
[INFO]

[INFO] --- spring-boot-maven-plugin:2.7.0:repackage (build-app) @ springboot-app-
example ---
[INFO] Attaching repackaged archive .../target/springboot-app-example-6.0.1-SNAPSHOT-
bootexec.jar with classifier bootexec ②
```

① standard Java JAR is built by the `maven-jar-plugin`

② standard Java JAR is augmented by the `spring-boot-maven-plugin`

6.2. Java MANIFEST.MF properties

The `spring-boot-maven-plugin` augmented the standard JAR by adding a few properties to the MANIFEST.MF file

```
$ unzip -qc target/springboot-app-example-6.0.1-SNAPSHOT-bootexec.jar META-
INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: Maven JAR Plugin 3.2.2
Build-Jdk-Spec: 17
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: info.ejava.examples.app.build.springboot.SpringBootApp
Spring-Boot-Version: 2.7.0
```

```
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx
Spring-Boot-Layers-Index: BOOT-INF/layers.idx
```

① `Main-Class` was set to a Spring Boot launcher

② `Start-Class` was set to the class we defined with @SpringBootApplication

6.3. JAR size

Notice that the size of the Spring Boot executable JAR is significantly larger the original standard JAR.

```
$ ls -lh target/*jar* | grep -v sources | cut -d\  -f9-99
8.4M Aug 28 15:19 target/springboot-app-example-6.0.1-SNAPSHOT-bootexec.jar ②
4.1K Aug 28 15:19 target/springboot-app-example-6.0.1-SNAPSHOT.jar  ①
```

① The original Java JAR with Spring Boot annotations was 4.1KB

② The Spring Boot JAR is 8.4MB

6.4. JAR Contents

Ref: spring.io Appendix E. The Executable Jar Format

Unlike WARs, a standard Java JAR does not provide a standard way to embed dependency JARs. Common approaches to embed dependencies within a single JAR include a "shaded" JAR where all dependency JAR are unwound and packaged as a single "uber" JAR

- positives
  - works
  - follows standard Java JAR constructs
- negatives
  - obscures contents of the application
  - problem if multiple source JARs use files with same path/name

Spring Boot creates a custom WAR-like structure

```
BOOT-INF/classes/info/ejava/examples/app/build/springboot/AppCommand.class
BOOT-INF/classes/info/ejava/examples/app/build/springboot/SpringBootApp.class ③
BOOT-INF/lib/javax.annotation-api-1.3.2.jar ②
...
BOOT-INF/lib/spring-boot-2.7.0.jar
BOOT-INF/lib/spring-context-5.3.20.jar
BOOT-INF/lib/spring-beans-5.3.20.jar
BOOT-INF/lib/spring-core-5.3.20.jar
```

```
...
META-INF/MANIFEST.MF
META-INF/maven/info.ejava.examples.app/springboot-app-example/pom.properties
META-INF/maven/info.ejava.examples.app/springboot-app-example/pom.xml
org/springframework/boot/loader/ExecutableArchiveLauncher.class ①
org/springframework/boot/loader/JarLauncher.class
...
org/springframework/boot/loader/util/SystemPropertyUtils.class
```

① Spring Boot loader classes hosted at the root `/`

② Local application classes hosted in `/BOOT-INF/classes`

③ Dependency JARs hosted in `/BOOT-INF/lib`

> Spring Boot can also use a standard WAR structure — to be deployed to a web server.
>
> - 99% of it is a standard WAR
>   - `/WEB-INF/classes`
>   - `/WEB-INF/lib`
> - Spring Boot loader classes hosted at the root `/`
> - Special directory for dependencies only used for non-container deployment
>   - `/WEB-INF/lib-provided`

6.5. Execute Command Line

```
springboot-app-example$ java -jar target/springboot-app-example-6.0.1-SNAPSHOT-
bootexec.jar ①
Running SpringApplication ②


  .   ____          _            __ _ _ ③
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.7.0})

2019-12-04 09:01:03.014  INFO 1287 --- [main] i.e.e.a.build.springboot.SpringBootApp:
\
  Starting SpringBootApp on Jamess-MBP with PID 1287 (.../springboot-app-
example/target/springboot-app-example-6.0.1-SNAPSHOT.jar \
  started by jim in .../springboot-app-example)
2019-12-04 09:01:03.017  INFO 1287 --- [main] i.e.e.a.build.springboot.SpringBootApp:
\
  No active profile set, falling back to default profiles: default
2019-12-04 09:01:03.416  INFO 1287 --- [main] i.e.e.a.build.springboot.SpringBootApp:
```

```
\
  Started SpringBootApp in 0.745 seconds (JVM running for 1.13)
Done SpringApplication ④
```

① Execute the JAR using the `java -jar` command

② Main executes and passes control to SpringApplication

③ Spring Boot bootstrap is started

④ SpringApplication terminates and returns control to our `main()`

Chapter 7. Add a Component to Output Message and Args

We have a lot of capability embedded into our current Spring Boot executable JAR that is there to bootstrap the application by looking around for components to activate. Lets explore this capability with a simple class that will take over the responsibility for the output of a message with the arguments to the program.

We want this class found by Spring's application startup processing, so we will:

```
// AppCommand.java
package info.ejava.examples.app.build.springboot; ②

import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import java.util.List;

@Component ①
public class AppCommand implements CommandLineRunner {
    public void run(String... args) throws Exception {
        System.out.println("Component code says Hello " + List.of(args));
    }
}
```

① Add a @Component annotation on the class

② Place the class in a Java package configured to be scanned

7.1. @Component Annotation

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class AppCommand implements CommandLineRunner {
```

Classes can be configured to have their instances managed by Spring. Class annotations can be used to express the purpose of a class and to trigger Spring into managing them in specific ways. The most generic form of component annotation is `@Component`. Others will include `@Repository`, `@Controller`, etc. Classes directly annotated with a `@Component` (or other annotation) indicates that Spring can instantiate instances of this class with no additional assistance from a `@Bean` factory.

7.2. Interface: CommandLineRunner

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
@Component
public class AppCommand implements CommandLineRunner {
    public void run(String... args) throws Exception {
    }
}
```

- Components implementing CommandLineRunner interface get called after application initialization
- Program arguments are passed to the `run()` method
- Can be used to perform one-time initialization at start-up
- Alternative Interface: ApplicationRunner
  - Components implementing ApplicationRunner are also called after application initialization
  - Program arguments are passed to its `run()` method have been wrapped in ApplicationArguments convenience class

Component startup can be ordered with the @Ordered Annotation.

7.3. @ComponentScan Tree

By default, the `@SpringBootApplication` annotation configured Spring to look at and below the Java package for our SpringBootApp class. I chose to place this component class in the same Java package as the application class

```
@SpringBootApplication
//  @ComponentScan
//  @SpringBootConfiguration
//  @EnableAutoConfiguration
public class SpringBootApp {
}
```

```
src/main/java
`-- info
    `-- ejava
        `-- springboot
            `-- examples
                `-- app
                    |-- AppCommand.java
                    `-- SpringBootApp.java
```

Chapter 8. Running the Spring Boot Application

```
$ java -jar target/springboot-app-example-6.0.1-SNAPSHOT-bootexec.jar

Running SpringApplication    ①

  .   ____          _            __ _ _ ②
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.7.0)

2019-09-06 15:56:45.666  INFO 11480 --- [           main]
i.e.s.examples.app.SpringBootApp
  : Starting SpringBootApp on Jamess-MacBook-Pro.local with PID 11480
(.../target/springboot-app-example-6.0.1-SNAPSHOT.jar ...)
2019-09-06 15:56:45.668  INFO 11480 --- [           main]
i.e.s.examples.app.SpringBootApp
  : No active profile set, falling back to default profiles: default
2019-09-06 15:56:46.146  INFO 11480 --- [           main]
i.e.s.examples.app.SpringBootApp
  : Started SpringBootApp in 5.791 seconds (JVM running for 6.161) ③
Hello                        ④ ⑤
Done SpringApplication       ⑥
```

① Our `SpringBootApp.main()` is called and logs `Running SpringApplication`

② `SpringApplication.run()` is called to execute the Spring Boot application

③ Our `AppCommand` component is found within the classpath at or under the package declaring `@SpringBootApplication`

④ The `AppCommand` component `run()` method is called and it prints out a message

⑤ The Spring Boot application terminates

⑥ Our `SpringBootApp.main()` logs `Done SpringApplication` an exits

8.1. Implementation Note

I added print statements directly in the Spring Boot Application's main() method to help illustrate when calls were made. This output could have been packaged into listener callbacks to leave the `main()` method implementation free — except to register the callbacks. If you happen to need more complex behavior to fire before the Spring context begins initialization, then look to add listeners of the SpringApplication instead.

Chapter 9. Configure pom.xml to Test

At this point we are again ready to setup an automated execution of our JAR as a part of the build. We can do that by adding a separate goal execution of the `spring-boot-maven-plugin`.

```
<build>
   ...
  <plugins>
      <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
          <executions>
              <execution>
                  <id>run-application</id> ①
                  <phase>integration-test</phase>
                  <goals>
                      <goal>run</goal>
                  </goals>
                  <configuration> ②
                      <arguments>Maven,plugin-supplied,args</arguments>
                  </configuration>
              </execution>
          </executions>
      </plugin>
  </plugins>
</build>
```

① new execution of the `run` goal to be performed during the Maven `integration-test` phase

② command line arguments passed to main

- Phase order
  1. …
  2. `package`
  3. `pre-integration`
  4. `integration-test`
  5. `post-integration`
  6. `verify`
  7. …

9.1. Execute JAR as part of the build

```
$ mvn clean verify
[INFO] Scanning for projects...
...
[INFO] --- spring-boot-maven-plugin:2.7.0:run (run-application) @ springboot-app-
```

```
example ---
[INFO] Attaching agents: [] ①
Running SpringApplication

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.0)

2022-07-02 14:11:46.110  INFO 48432 --- [           main]
i.e.e.a.build.springboot.SpringBootApp   : Starting SpringBootApp using Java 17.0.3 on
Jamess-MacBook-Pro.local with PID 48432 (.../springboot-app-example/target/classes
started by jim in .../springboot-app-example)
2022-07-02 14:11:46.112  INFO 48432 --- [           main]
i.e.e.a.build.springboot.SpringBootApp   : No active profile set, falling back to 1
default profile: "default"
2022-07-02 14:11:46.463  INFO 48432 --- [           main]
i.e.e.a.build.springboot.SpringBootApp   : Started SpringBootApp in 0.611 seconds (JVM
running for 0.87)
Component code says Hello [Maven, plugin-supplied, args] ②
Done SpringApplication
```

① Our plugin is executing

② Our application was executed and the results displayed

Chapter 10. Summary

As a part of this material, the student has learned how to:

1. Add Spring Boot constructs and artifact dependencies to the Maven POM
2. Define Application class with a main() method
3. Annotate the application class with @SpringBootApplication (and optionally use lower-level annotations)
4. Place the application class in a Java package that is at or above the Java packages with beans that will make-up the core of your application
5. Add component classes that are core to your application to your Maven module
6. Typically define components in a Java package that is at or below the Java package for the `SpringBootApplication`
7. Annotate components with @Component (or other special-purpose annotations used by Spring)
8. Execute application like a normal executable JAR