**Spring Boot**

Spring Boot helps you to create stand-alone, production-grade Spring-based applications that you can run. We take an opinionated view of the Spring platform and third-party libraries, so that you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

You can use Spring Boot to create Java applications that can be started by using java -jar or more traditional war deployments.

**primary goals are:**

1. Faster development with starter projects
2. Default configuration with help of auto-configuration.
3. Provides Common set of features used by most of the projects such as embedded servers, security, metrics, health checks, and externalized configuration.
4. Absolutely no code generation (when not targeting native image) and no requirement for XML configuration.

| **Build Tool** | **Version** |
| --- | --- |
| Maven | 3.6.3 or later |
| Gradle | 7.x (7.5 or later) and 8.x |

**Servlet Containers**

Spring Boot supports the following embedded servlet containers:

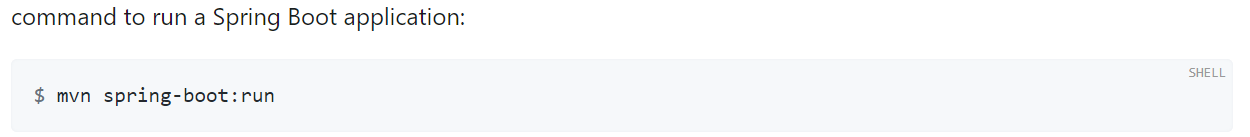
| **Name** | **Servlet Version** |
| --- | --- |
| Tomcat 10.1 | 6.0 |
| Jetty 12.0 | 6.0 |
| Undertow 2.3 | 6.0 |

**Setting up the project with Maven**

A computer screen shot of a program code

Description automatically generated

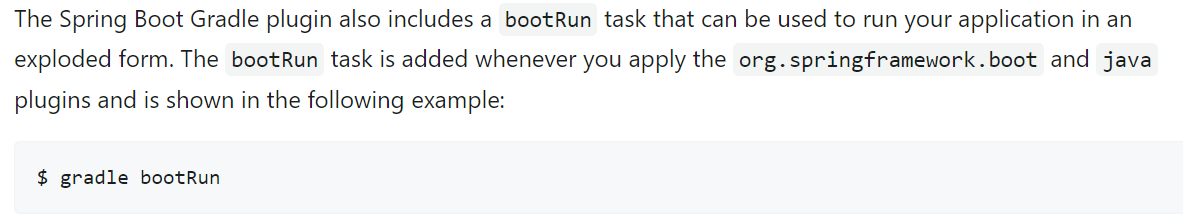
$ mvn dependency:tree -> to see the dependency tree



**Setting up the project with Gradle**



gradle dependencies -> to see the dependency tree



**First Application**



* The @RestController annotation is not considered a stereotype annotation. It is a specialized version of the @Controller annotation and is a combination of the @Controller and @ResponseBody annotations.
* The @ResponseBody annotation in Spring Boot is used to tell Spring that the object returned by a controller method should be written to the response body instead of being rendered as a view. This is useful when you want to return JSON, XML, or other data formats that are not HTML.
* The @RestController and @RequestMapping annotations are Spring MVC annotations (they are not specific to Spring Boot)
* Stereotype annotations in Spring Boot are used to provide additional information about a class to the Spring container. This information can be used by the Spring container to automatically configure the class and its dependencies.

**The @SpringBootApplication Annotation**

* The second class-level annotation is @SpringBootApplication. This annotation is known as a meta-annotation, it combines @SpringBootConfiguration, @EnableAutoConfiguration and @ComponentScan.
* Of those, the annotation we’re most interested in here is @EnableAutoConfiguration. @EnableAutoConfiguration tells Spring Boot to “guess” how you want to configure Spring, based on the jar dependencies that you have added. Since spring-boot-starter-web added Tomcat and Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

**The “main” Method**

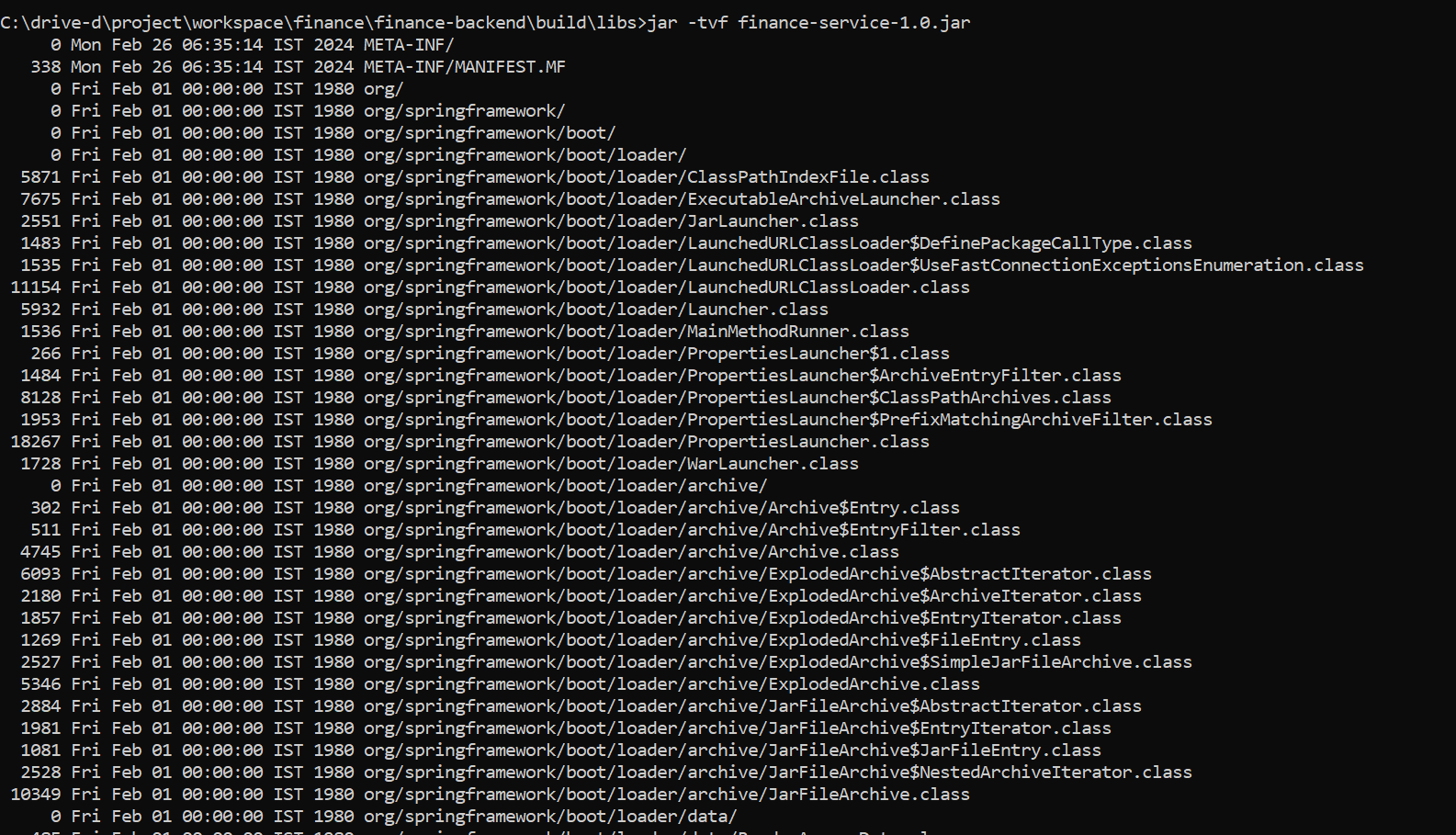
* The final part of our application is the main method. This is a standard method that follows the Java convention for an application entry point. Our main method delegates to Spring Boot’s SpringApplication class by calling run. SpringApplication bootstraps our application, starting Spring, which, in turn, starts the auto-configured Tomcat web server. We need to pass MyApplication.class as an argument to the run method to tell SpringApplication which is the primary Spring component. The args array is also passed through to expose any command-line arguments.

**To Create Executable Jar for SpringBootApplication**

mvn package will generate jar in target directory.

To view the content of use below command

$ jar tvf target/myproject-0.0.1-SNAPSHOT.jar





**Upgrading to a New Feature Release**

When upgrading to a new feature release, some properties may have been renamed or removed. Spring Boot provides a way to analyse your application’s environment and print diagnostics at startup, but also temporarily migrate properties at runtime for you. To enable that feature, add the following dependency to your project:

<dependency>

<groupId>org.springframework.boot</groupId>

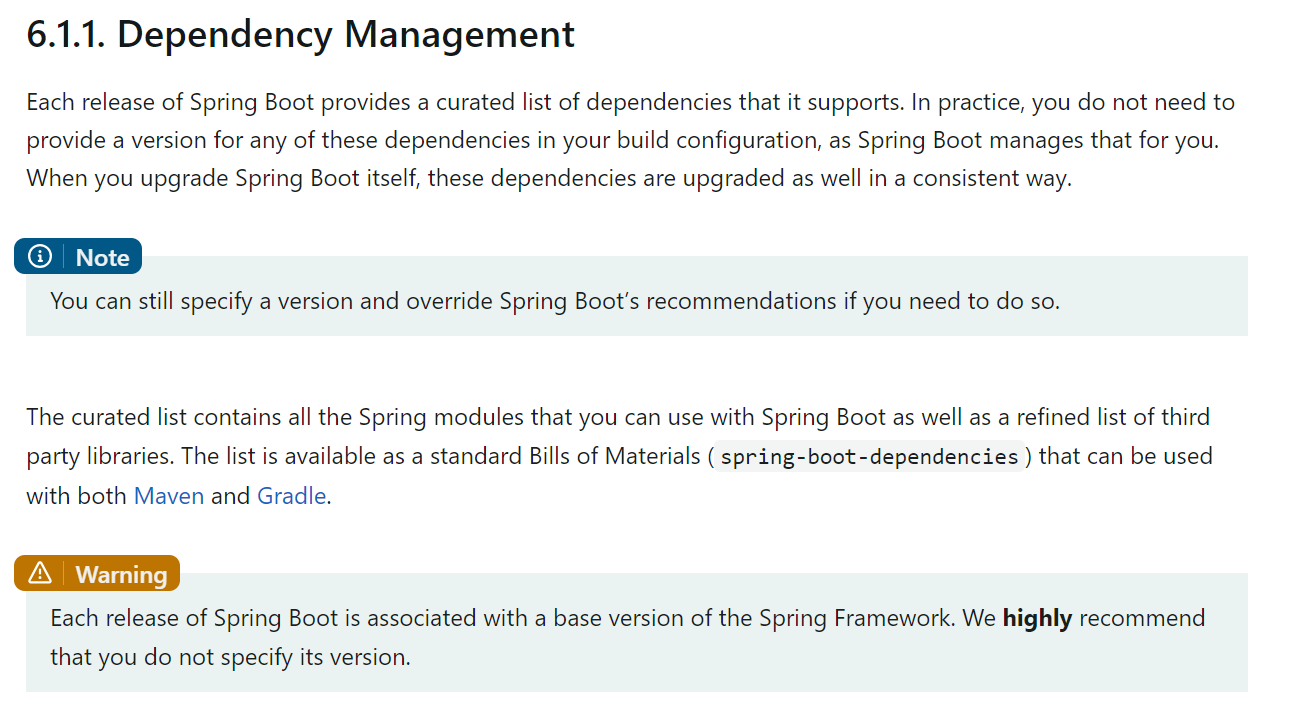
<artifactId>spring-boot-properties-migrator</artifactId>

<scope>runtime</scope>

</dependency>

Properties that are added late to the environment, such as when using @PropertySource, will not be considered.

Once you finish the migration, please make sure to remove this module from your project’s dependencies.



**Starters**

* Starters are a set of convenient dependency descriptors that you can include in your application. You get a one-stop shop for all the Spring and related technologies that you need without having to hunt through sample code and copy-paste loads of dependency descriptors. For example, if you want to get started using Spring and JPA for database access, include the spring-boot-starter-data-jpa dependency in your project.
* The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.
* All official starters follow a similar naming pattern; spring-boot-starter-\*, where \* is a particular type of application.
* third party starters should not start with spring-boot, as it is reserved for official Spring Boot artifacts. Rather, a third-party starter typically starts with the name of the project. For example, a third-party starter project called thirdpartyproject would typically be named thirdpartyproject-spring-boot-starter.

**Here are some of the differences between Spring Boot 2 and Spring Boot 3:**

* **Java Version:**

Spring Boot 2 requires Java 11 or higher, while Spring Boot 3 requires Java 17 or higher.

* **Jakarta EE:**

Spring Boot 2 uses Java EE APIs, while Spring Boot 3 uses Jakarta EE APIs. Jakarta EE is the new name for Java EE, and it is a set of Java APIs that provide enterprise features such as web services, transaction management, and security.

* **Web Server:**

Spring Boot 2 uses Tomcat as its default web server, while Spring Boot 3 uses Undertow as its default web server. Undertow is a lightweight web server that is designed for high performance.

* **Reactive Programming:**

Spring Boot 3 includes support for reactive programming, which is a programming paradigm that is based on asynchronous data streams. Reactive programming can be used to build applications that are more scalable and responsive.

* **Spring Framework:**

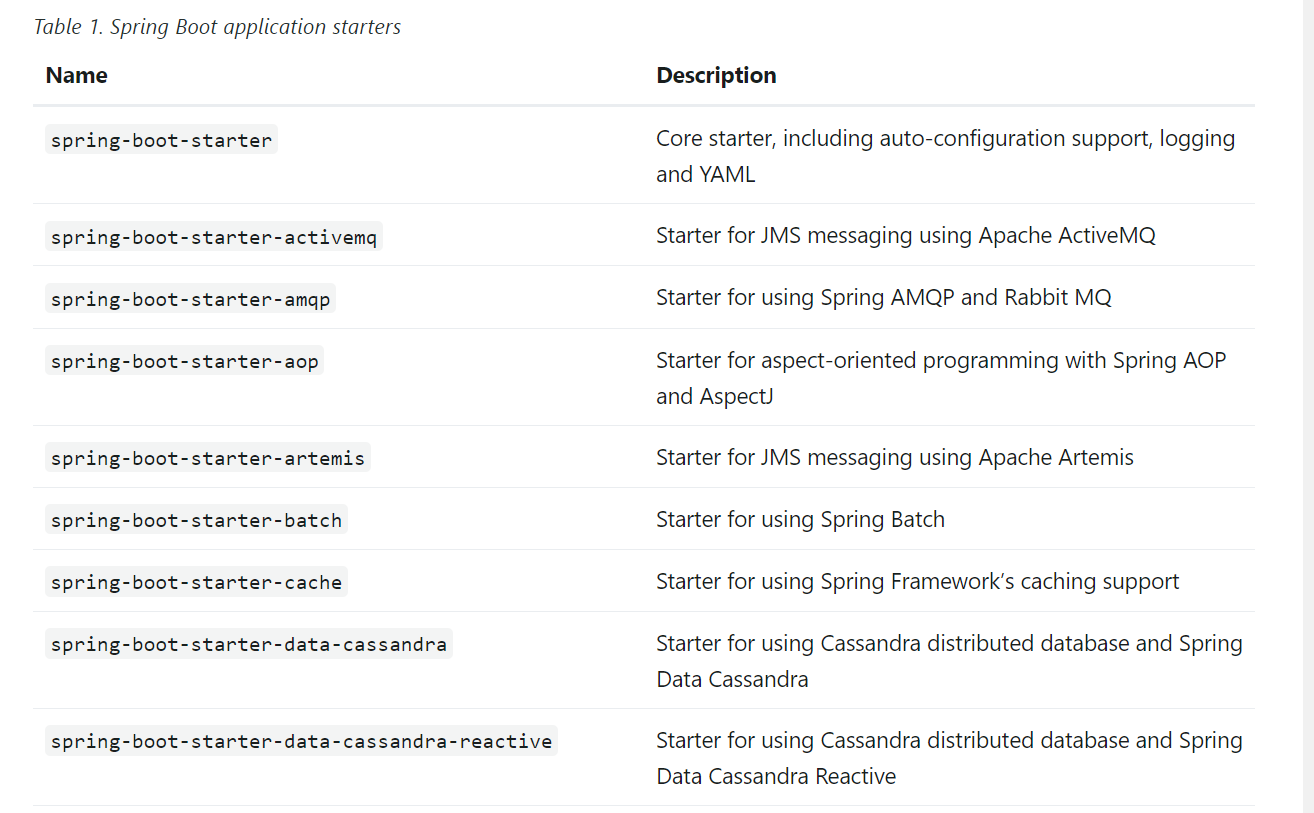
Spring Boot 2 uses Spring Framework 5, while Spring Boot 3 uses Spring Framework 6. Spring Framework 6 includes a number of new features and improvements, such as support for reactive programming and Jakarta EE.

* **GraalVM Native Image Support:**

Spring Boot 3 includes support for GraalVM native images. GraalVM native images are ahead-of-time compiled images that can be used to start up Spring Boot applications much faster. GraalVM can create a native image that doesn't require a JVM to run. GraalVM is a Java Virtual Machine (JVM) implementation that usesAhead-Of-Time (AOT) compilation to create native executables for Java applications

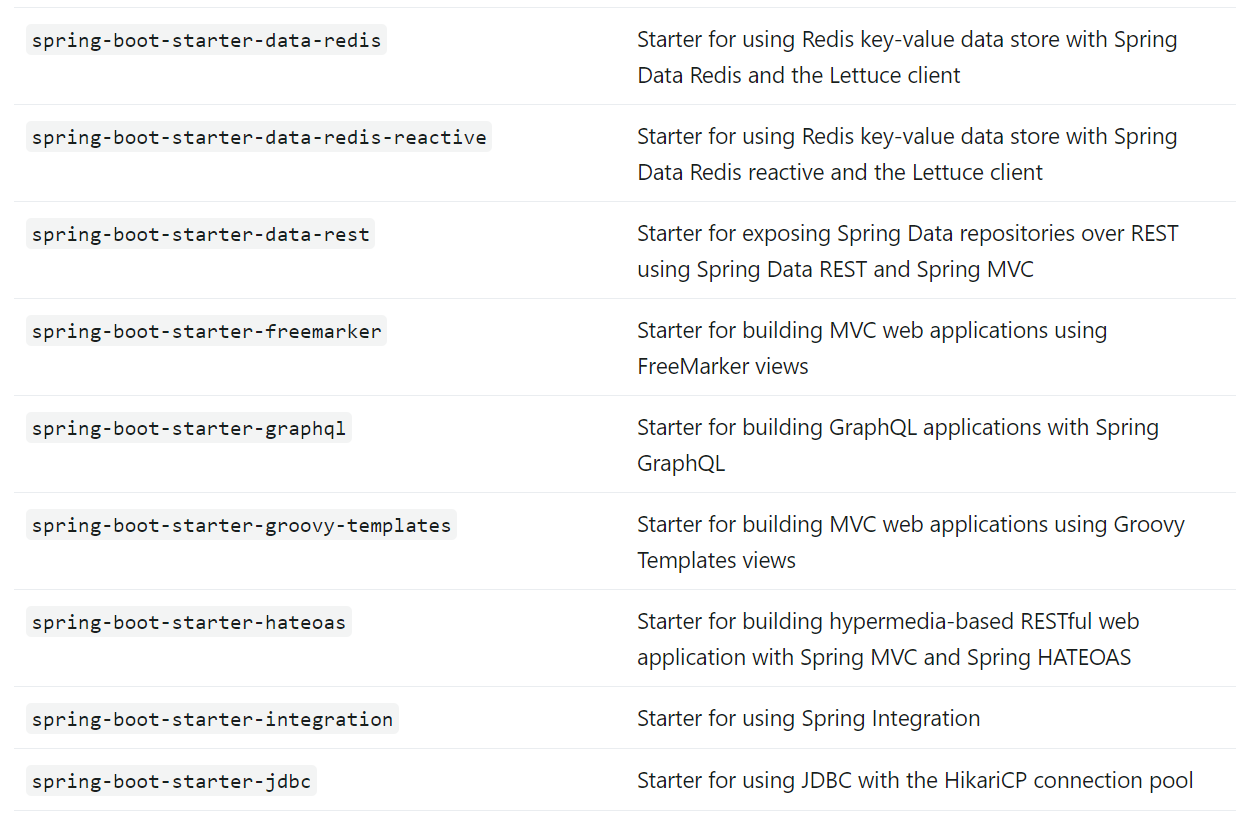
* **Improved Observability:**

Spring Boot 3 includes improved observability features, such as support for **Micrometer** and **Micrometer Tracing**. Micrometer and Micrometer Tracing can be used to collect and analyze metrics about your Spring Boot application.



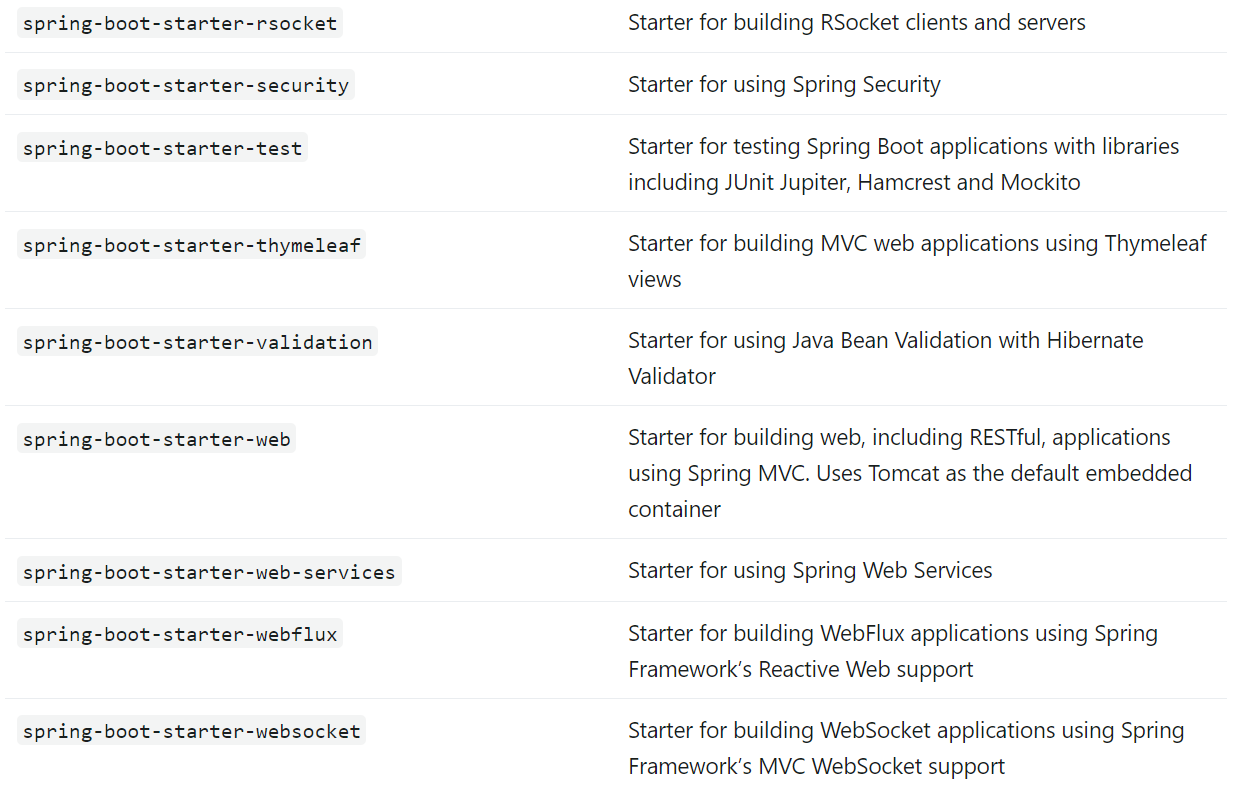
A screenshot of a computer

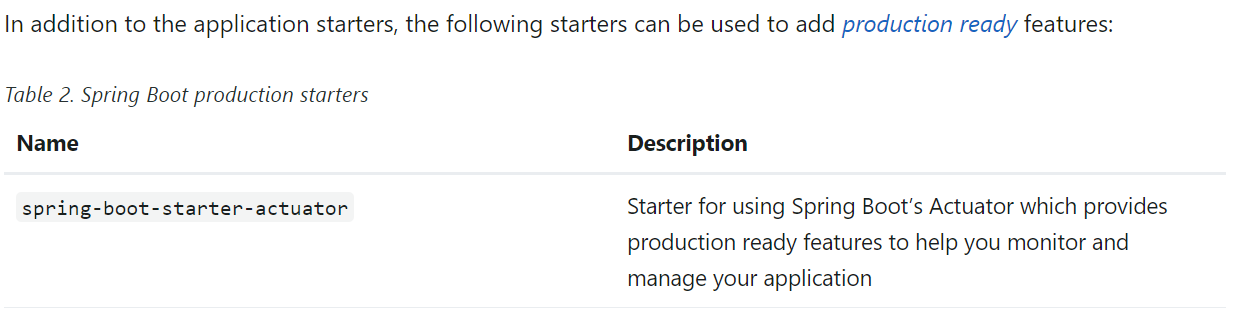
Description automatically generated



A screenshot of a computer

Description automatically generated





A screenshot of a computer

Description automatically generated

**Using the “default” Package**

When a class does not include a package declaration, it is considered to be in the “default package”. The use of the “default package” is generally discouraged and should be avoided. It can cause particular problems for Spring Boot applications that use the @ComponentScan, @ConfigurationPropertiesScan, @EntityScan, or @SpringBootApplication annotations, since every class from every jar is read.

**@ConfigurationPropertiesScan**

Spring finds and registers @ConfigurationProperties classes via classpath scanning. Scanning of @ConfigurationProperties needs to be explicitly opted into by adding the @ConfigurationPropertiesScan annotation. Therefore, we don’t have to annotate such classes with @Component (and other meta-annotations like @Configuration), or even use the @EnableConfigurationProperties.

The classpath scanner enabled by @SpringBootApplication finds the ConfigProperties class, even though we didn’t annotate this class with @Component.

In addition, we can use the @ConfigurationPropertiesScan annotation to scan custom locations for configuration property classes.

A computer code with black text

Description automatically generated

**@EntityScan**

When writing our Spring application we will usually have entity classes – those annotated with @Entity annotation. We can consider two approaches to placing our entity classes:

* Under the application main package or its sub-packages
* Use a completely different root package

In the first scenario, we could use @EnableAutoConfiguration to enable Spring to auto-configure the application context.

In the second scenario, we would provide our application with the information where these packages could be found. For this purpose, we would use @EntityScan.

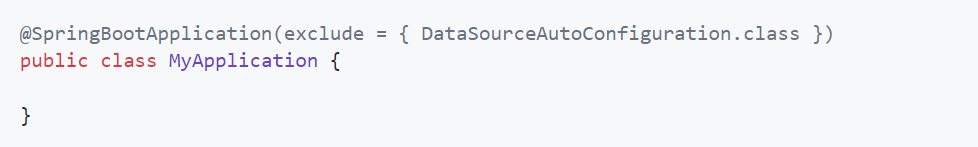
**@EntityScan vs. @ComponentScan**

In the end, we can say that these two annotations are intended for completely different purposes.

Their similarity is that they both contribute to our Spring application configuration. @EntityScan should specify which packages do we want to scan for entity classes. On the other hand, @ComponentScan is a choice when specifying which packages should be scanned for Spring beans.

**Disabling Specific Auto-configuration Classes**

If you find that specific auto-configuration classes that you do not want are being applied, you can use the exclude attribute of @SpringBootApplication to disable them, as shown in the following example:



**Spring Beans and Dependency Injection**

generally recommend using constructor injection to wire up dependencies and @ComponentScan to find beans.

If you structure your code as suggested above (locating your application class in a top package), you can add @ComponentScan without any arguments or use the @SpringBootApplication annotation which implicitly includes it. All of your application components (@Component, @Service, @Repository, @Controller, and others) are automatically registered as Spring Beans.

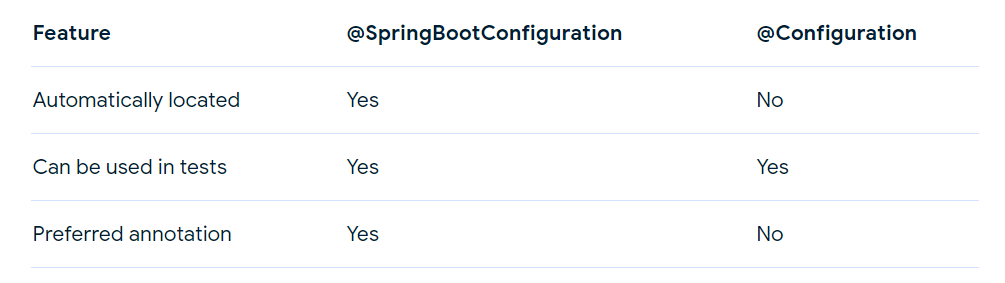
**@SpringBootApplication Annotation**

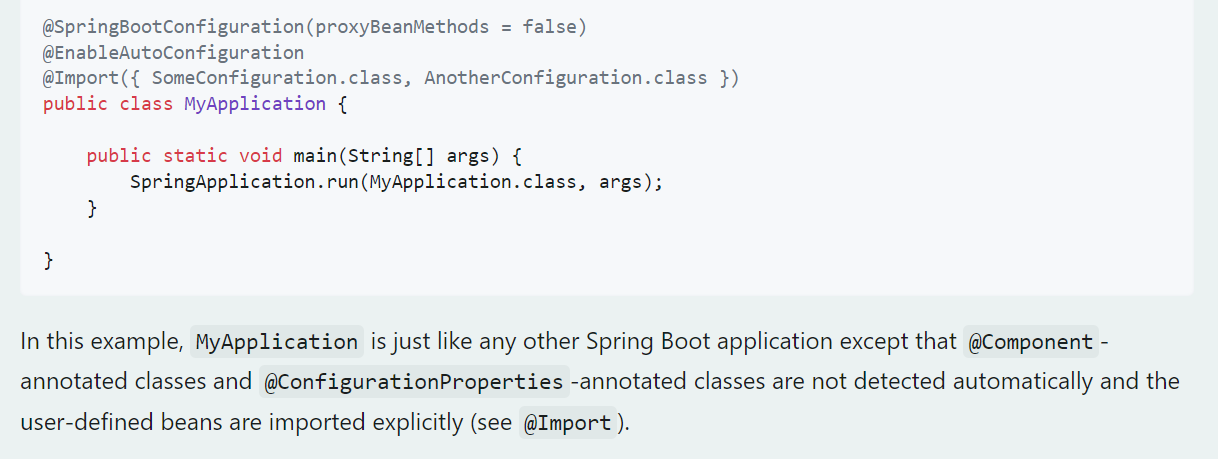
Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single @SpringBootApplication annotation can be used to enable those three features, that is:

* @EnableAutoConfiguration: enable [Spring Boot’s auto-configuration mechanism](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.auto-configuration)
* @ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.structuring-your-code))
* @SpringBootConfiguration: enable registration of extra beans in the context or the import of additional configuration classes. An alternative to Spring’s standard @Configuration that aids [configuration detection](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.testing.spring-boot-applications.detecting-configuration) in your integration tests.

**@SpringBootConfiguration VS @Configuration**

The main difference between the two annotations is that @SpringBootConfiguration allows the configuration to be automatically located. This can be especially useful for unit or integration tests. In general, the @SpringBootConfiguration annotation is preferred over the @Configuration annotation, unless you need to explicitly specify the location of the configuration.





**Externalized Configuration**

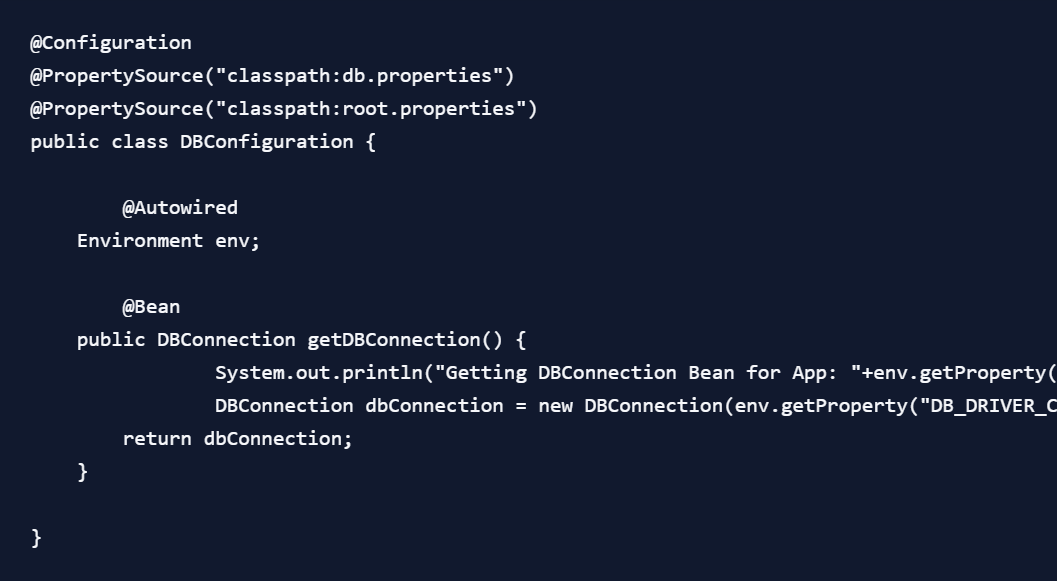
* Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources including Java properties files, YAML files, environment variables, and command-line arguments.
* Property values can be injected directly into your beans by using the @Value annotation, accessed through Spring’s Environment abstraction, or be bound to structured objects through @ConfigurationProperties.
* Spring Boot uses a very particular PropertySource order that is designed to allow sensible overriding of values. Later property sources can override the values defined in earlier ones. Sources are considered in the following order:

1. Default properties (specified by setting **SpringApplication.setDefaultProperties**).
2. [@PropertySource](https://docs.spring.io/spring-framework/docs/6.1.4/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotations on your @Configuration classes. Please note that such property sources are not added to the Environment until the application context is being refreshed. This is too late to configure certain properties such as logging.\* and spring.main.\* which are read before refresh begins.
3. Config data (such as application.properties files).
4. A **RandomValuePropertySource** that has properties only in random.\*.
5. OS environment variables.
6. Java System properties (System.getProperties()).
7. JNDI attributes from java:comp/env.
8. ServletContext init parameters.
9. ServletConfig init parameters.
10. Properties from SPRING\_APPLICATION\_JSON (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. properties attribute on your tests. Available on [@SpringBootTest](https://docs.spring.io/spring-boot/docs/3.2.3/api/org/springframework/boot/test/context/SpringBootTest.html) and the [test annotations for testing a particular slice of your application](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.testing.spring-boot-applications.autoconfigured-tests).
13. [@DynamicPropertySource](https://docs.spring.io/spring-framework/docs/6.1.4/javadoc-api/org/springframework/test/context/DynamicPropertySource.html) annotations in your tests.
14. [@TestPropertySource](https://docs.spring.io/spring-framework/docs/6.1.4/javadoc-api/org/springframework/test/context/TestPropertySource.html) annotations on your tests.
15. [Devtools global settings properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.devtools.globalsettings) in the $HOME/.config/spring-boot directory when devtools is active.



**@PropertySource**

Spring @PropertySource annotation is used to provide properties file to Spring Environment. This annotation is used with @Configuration classes. Spring PropertySource annotation is repeatable, means you can have multiple PropertySource on a Configuration class. This feature is available if you are using Java 8 or higher version.





**Accessing Command Line Properties**

* By default, SpringApplication converts any command line option arguments (that is, arguments starting with --, such as --server.port=9000) to a property and adds them to the Spring Environment. As mentioned previously, command line properties always take precedence over file-based property sources.
* If you do not want command line properties to be added to the Environment, you can disable them by using SpringApplication.setAddCommandLineProperties(false).

**RandomValuePropertySource**

RandomValuePropertySource is a PropertySource in Spring Boot that returns a random value for any property that starts with "random.". It is automatically applied when the application finds property values of the format "${random.xyz..}".



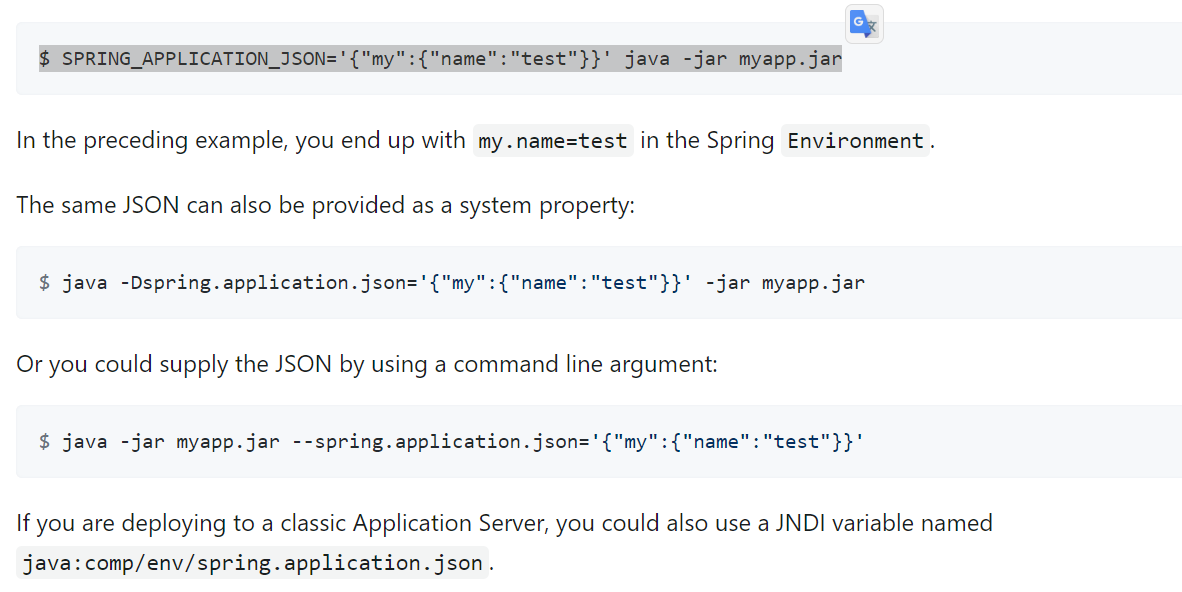
Here is an example of how to use RandomValuePropertySource:

@SpringBootApplication  
public class MyApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(MyApplication.class, args);  
 }  
  
 @Value("${random.int}")  
 private int randomInt;  
  
 @Value("${random.long}")  
 private long randomLong;  
  
 @Value("${random.uuid}")  
 private UUID randomUUID;  
  
 @Value("${random.bytes}")  
 private byte[] randomBytes;  
  
 *// ...*  
}

In this example, the application will inject a random integer, long, UUID, and byte array into the **MyApplication** class. The values of these properties will be different each time the application is run.

**JSON Application Properties**

* Spring Boot allows you to encode a block of properties into a single JSON structure.
* When your application starts, any **spring.application.json** or **SPRING\_APPLICATION\_JSON** properties will be parsed and added to the Environment.
* For example, the **SPRING\_APPLICATION\_JSON** property can be supplied on the command line in a UN\*X shell as an environment variable:



#### External Application Properties

You can place your application.properties in one of the following locations:

1. Under /config sub-directory of the current directory
2. The current directory
3. A classpath /config package
4. The classpath root

**Benefits of Devtools**

Spring Boot Devtools is a module that provides a number of features to improve the development experience of Spring Boot applications. These features include:

* Automatic restart:

When a source file is changed, the application will automatically restart. This can save a lot of time when developing and testing an application.

* LiveReload:

When a static resource file is changed, the browser will automatically reload the page. This can be very useful for seeing changes to CSS and JavaScript files without having to manually refresh the page.

* Remote debugging:

Devtools makes it possible to debug a Spring Boot application running on a remote server. This can be very useful for debugging applications that are deployed to production.

* H2 Console:

Devtools includes an embedded H2 database console. This can be very useful for viewing and managing data in an H2 database.

**GraalVM Native Image Support**

* GraalVM Native Images are standalone executables that can be generated by processing compiled Java applications ahead-of-time. Native Images generally have a smaller memory footprint and start faster than their JVM counterparts.
* GraalVM Native Images provide a new way to deploy and run Java applications. Compared to the Java Virtual Machine, native images can run with a smaller memory footprint and with much faster startup times.
* They are well suited to applications that are deployed using container images and are especially interesting when combined with "Function as a service" (FaaS) platforms.
* Unlike traditional applications written for the JVM, GraalVM Native Image applications require ahead-of-time processing in order to create an executable. This ahead-of-time processing involves statically analyzing your application code from its main entry point.]
* A GraalVM Native Image is a complete, platform-specific executable. You do not need to ship a Java Virtual Machine in order to run a native image.

@Transactional(propagation=Propagation.REQUIRED)  
def outer\_transaction():  
 *# Do some work*  
  
 *# Call the read-only transaction*  
 inner\_transaction()  
  
@Transactional(propagation=Propagation.REQUIRES\_NEW, readOnly=True)  
def inner\_transaction():  
 # Do some read-only work

Use code with caution.

[Learn more](https://support.google.com/legal/answer/13505487?hl=en-IN)

In this example, the outer\_transaction() method will start a new transaction. The inner\_transaction() method will then be called within the context of the outer transaction. The inner\_transaction() method will start a new nested transaction that is read-only.

The inner\_transaction() method can perform any read-only operations that it needs to perform. The changes made in the inner\_transaction() method will not be committed to the database until the outer transaction commits.

If the outer transaction rolls back, the changes made in the inner\_transaction() method will also be rolled back.

##### Relaxed Binding

Spring Boot uses some relaxed rules for binding Environment properties to @ConfigurationProperties beans, so there does not need to be an exact match between the Environment property name and the bean property name. Common examples where this is useful include dash-separated environment properties (for example, context-path binds to contextPath), and capitalized environment properties (for example, PORT binds to port).

As an example, consider the following @ConfigurationProperties class:

**Java**

**Kotlin**

@ConfigurationProperties(prefix = "my.main-project.person")

public class MyPersonProperties {

private String firstName;

public String getFirstName() {

return this.firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

}

With the preceding code, the following properties names can all be used:

| *Table 6. relaxed binding* | |
| --- | --- |
| **Property** | **Note** |
| my.main-project.person.first-name | Kebab case, which is recommended for use in .properties and YAML files. |
| my.main-project.person.firstName | Standard camel case syntax. |
| my.main-project.person.first\_name | Underscore notation, which is an alternative format for use in .properties and YAML files. |
| MY\_MAINPROJECT\_PERSON\_FIRSTNAME | Upper case format, which is recommended when using system environment variables. |