# Part III. Using Spring Boot

This section goes into more detail about how you should use Spring Boot. It covers topics such as build systems, auto-configuration, and how to run your applications. We also cover some Spring Boot best practices. Although there is nothing particularly special about Spring Boot (it is just another library that you can consume), there are a few recommendations that, when followed, make your development process a little easier.

If you are starting out with Spring Boot, you should probably read the *Getting Started* guide before diving into this section.

# 13. Build Systems

It is strongly recommended that you choose a build system that supports *dependency management* and that can consume artifacts published to the “Maven Central” repository. We would recommend that you choose Maven or Gradle. It is possible to get Spring Boot to work with other build systems (Ant, for example), but they are not particularly well supported.

## 13.1 Dependency Management

Each release of Spring Boot provides a curated list of dependencies that it supports. In practice, you do not need to provide a version for any of these dependencies in your build configuration, as Spring Boot manages that for you. When you upgrade Spring Boot itself, these dependencies are upgraded as well in a consistent way.

> **Note**
>
> You can still specify a version and override Spring Boot’s recommendations if you need to do so.

The curated list contains all the spring modules that you can use with Spring Boot as well as a refined list of third party libraries. The list is available as a standard Bills of Materials (`spring-boot-dependencies`) that can be used with both Maven and Gradle.

> **Warning**
>
> Each release of Spring Boot is associated with a base version of the Spring Framework. We **highly** recommend that you not specify its version.

## 13.2 Maven

Maven users can inherit from the `spring-boot-starter-parent` project to obtain sensible defaults. The parent project provides the following features:

- Java 1.8 as the default compiler level.
- UTF-8 source encoding.
- A Dependency Management section, inherited from the spring-boot-dependencies pom, that manages the versions of common dependencies. This dependency management lets you omit <version> tags for those dependencies when used in your own pom.
- Sensible resource filtering.
- Sensible plugin configuration (exec plugin, Git commit ID, and shade).
- Sensible resource filtering for `application.properties` and `application.yml` including profile-specific files (for example, `application-dev.properties` and `application-dev.yml`)

Note that, since the `application.properties` and `application.yml` files accept Spring style placeholders (`${…}`), the Maven filtering is changed to use `@..@` placeholders. (You can override that by setting a Maven property called `resource.delimiter`.)

## Inheriting the Starter Parent

To configure your project to inherit from the `spring-boot-starter-parent`, set the `parent` as follows:

```
<!-- Inherit defaults from Spring Boot -->
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>2.1.0.BUILD-SNAPSHOT</version>
</parent>
```

> **Note**
>
> You should need to specify only the Spring Boot version number on this dependency. If you import additional starters, you can safely omit the version number.

With that setup, you can also override individual dependencies by overriding a property in your own project. For instance, to upgrade to another Spring Data release train, you would add the following to your `pom.xml`:

```
<properties>
 <spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version>
</properties>
```

> **Tip**
>
> Check the spring-boot-dependencies pom for a list of supported properties.

## Using Spring Boot without the Parent POM

Not everyone likes inheriting from the `spring-boot-starter-parent` POM. You may have your own corporate standard parent that you need to use or you may prefer to explicitly declare all your Maven configuration.

If you do not want to use the `spring-boot-starter-parent`, you can still keep the benefit of the dependency management (but not the plugin management) by using a `scope=import` dependency, as follows:

```
<dependencyManagement>
  <dependencies>
  <dependency>
   <!-- Import dependency management from Spring Boot -->
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-dependencies</artifactId>
   <version>2.1.0.BUILD-SNAPSHOT</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

The preceding sample setup does not let you override individual dependencies by using a property, as explained above. To achieve the same result, you need to add an entry in the `dependencyManagement` of your project **before** the `spring-boot-dependencies` entry. For instance, to upgrade to another Spring Data release train, you could add the following element to your `pom.xml`:

```
<dependencyManagement>
```

```
 <dependencies>
  <!-- Override Spring Data release train provided by Spring Boot -->
  <dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-releasetrain</artifactId>
   <version>Fowler-SR2</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-dependencies</artifactId>
   <version>2.1.0.BUILD-SNAPSHOT</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

**Note**

In the preceding example, we specify a *BOM*, but any dependency type can be overridden in the same way.

### Using the Spring Boot Maven Plugin

Spring Boot includes a Maven plugin that can package the project as an executable jar. Add the plugin to your `<plugins>` section if you want to use it, as shown in the following example:

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

**Note**

If you use the Spring Boot starter parent pom, you need to add only the plugin. There is no need to configure it unless you want to change the settings defined in the parent.

## 13.3 Gradle

To learn about using Spring Boot with Gradle, please refer to the documentation for Spring Boot's Gradle plugin:

- Reference (HTML and PDF)
- API

## 13.4 Ant

It is possible to build a Spring Boot project using Apache Ant+Ivy. The `spring-boot-antlib` "AntLib" module is also available to help Ant create executable jars.

To declare dependencies, a typical `ivy.xml` file looks something like the following example:

```
<ivy-module version="2.0">
 <info organisation="org.springframework.boot" module="spring-boot-sample-ant" />
 <configurations>
  <conf name="compile" description="everything needed to compile this module" />
  <conf name="runtime" extends="compile" description="everything needed to run this module" />
 </configurations>
 <dependencies>
  <dependency org="org.springframework.boot" name="spring-boot-starter"
   rev="${spring-boot.version}" conf="compile" />
 </dependencies>
</ivy-module>
```

A typical `build.xml` looks like the following example:

```
<project
 xmlns:ivy="antlib:org.apache.ivy.ant"
 xmlns:spring-boot="antlib:org.springframework.boot.ant"
 name="myapp" default="build">

 <property name="spring-boot.version" value="2.1.0.BUILD-SNAPSHOT" />

 <target name="resolve" description="--> retrieve dependencies with ivy">
  <ivy:retrieve pattern="lib/[conf]/[artifact]-[type]-[revision].[ext]" />
 </target>

 <target name="classpaths" depends="resolve">
  <path id="compile.classpath">
   <fileset dir="lib/compile" includes="*.jar" />
  </path>
 </target>

 <target name="init" depends="classpaths">
  <mkdir dir="build/classes" />
 </target>

 <target name="compile" depends="init" description="compile">
  <javac srcdir="src/main/java" destdir="build/classes" classpathref="compile.classpath" />
 </target>

 <target name="build" depends="compile">
  <spring-boot:exejar destfile="build/myapp.jar" classes="build/classes">
   <spring-boot:lib>
    <fileset dir="lib/runtime" />
   </spring-boot:lib>
  </spring-boot:exejar>
 </target>
</project>
```

**Tip**

If you do not want to use the `spring-boot-antlib` module, see the *Section 88.9, "Build an Executable Archive from Ant without Using `spring-boot-antlib`"* "How-to" .

## 13.5 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.

**What's in a name**

All **official** starters follow a similar naming pattern; `spring-boot-starter-*`, where `*` is a particular type of application. This naming structure is intended to help when you need to find a starter. The Maven integration in many IDEs lets you search dependencies by name. For example, with the appropriate Eclipse or STS plugin installed, you can press `ctrl-space` in the POM editor and type "spring-boot-starter" for a complete list.

As explained in the "Creating Your Own Starter" section, 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`.

The following application starters are provided by Spring Boot under the `org.springframework.boot` group:

*Table 13.1. Spring Boot application starters*

| Name | Description | Pom |
|---|---|---|
| `spring-boot-starter` | Core starter, including auto-configuration support, logging and YAML | Pom |
| `spring-boot-starter-activemq` | Starter for JMS messaging using Apache ActiveMQ | Pom |
| `spring-boot-starter-amqp` | Starter for using Spring AMQP and Rabbit MQ | Pom |
| `spring-boot-starter-aop` | Starter for aspect-oriented programming with Spring AOP and AspectJ | Pom |
| `spring-boot-starter-artemis` | Starter for JMS messaging using Apache Artemis | Pom |
| `spring-boot-starter-batch` | Starter for using Spring Batch | Pom |
| `spring-boot-starter-cache` | Starter for using Spring Framework's caching support | Pom |
| `spring-boot-starter-cloud-connectors` | Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku | Pom |
| `spring-boot-starter-data-cassandra` | Starter for using Cassandra distributed database and Spring Data Cassandra | Pom |

| Name | Description | Pom |
| --- | --- | --- |
| `spring-boot-starter-data-cassandra-reactive` | Starter for using Cassandra distributed database and Spring Data Cassandra Reactive | Pom |
| `spring-boot-starter-data-couchbase` | Starter for using Couchbase document-oriented database and Spring Data Couchbase | Pom |
| `spring-boot-starter-data-couchbase-reactive` | Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive | Pom |
| `spring-boot-starter-data-elasticsearch` | Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch | Pom |
| `spring-boot-starter-data-jpa` | Starter for using Spring Data JPA with Hibernate | Pom |
| `spring-boot-starter-data-ldap` | Starter for using Spring Data LDAP | Pom |
| `spring-boot-starter-data-mongodb` | Starter for using MongoDB document-oriented database and Spring Data MongoDB | Pom |
| `spring-boot-starter-data-mongodb-reactive` | Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive | Pom |
| `spring-boot-starter-data-neo4j` | Starter for using Neo4j graph database and Spring Data Neo4j | Pom |
| `spring-boot-starter-data-redis` | Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client | Pom |
| `spring-boot-starter-data-redis-reactive` | Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client | Pom |
| `spring-boot-starter-data-rest` | Starter for exposing Spring Data repositories over REST using Spring Data REST | Pom |

| Name | Description | Pom |
|---|---|---|
| spring-boot-starter-data-solr | Starter for using the Apache Solr search platform with Spring Data Solr | Pom |
| spring-boot-starter-freemarker | Starter for building MVC web applications using FreeMarker views | Pom |
| spring-boot-starter-groovy-templates | Starter for building MVC web applications using Groovy Templates views | Pom |
| spring-boot-starter-hateoas | Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS | Pom |
| spring-boot-starter-integration | Starter for using Spring Integration | Pom |
| spring-boot-starter-jdbc | Starter for using JDBC with the HikariCP connection pool | Pom |
| spring-boot-starter-jersey | Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web | Pom |
| spring-boot-starter-jooq | Starter for using jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc | Pom |
| spring-boot-starter-json | Starter for reading and writing json | Pom |
| spring-boot-starter-jta-atomikos | Starter for JTA transactions using Atomikos | Pom |
| spring-boot-starter-jta-bitronix | Starter for JTA transactions using Bitronix | Pom |
| spring-boot-starter-jta-narayana | Starter for JTA transactions using Narayana | Pom |
| spring-boot-starter-mail | Starter for using Java Mail and Spring Framework's email sending support | Pom |

| Name | Description | Pom |
|---|---|---|
| `spring-boot-starter-mustache` | Starter for building web applications using Mustache views | Pom |
| `spring-boot-starter-quartz` | Starter for using the Quartz scheduler | Pom |
| `spring-boot-starter-security` | Starter for using Spring Security | Pom |
| `spring-boot-starter-test` | Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito | Pom |
| `spring-boot-starter-thymeleaf` | Starter for building MVC web applications using Thymeleaf views | Pom |
| `spring-boot-starter-validation` | Starter for using Java Bean Validation with Hibernate Validator | Pom |
| `spring-boot-starter-web` | Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container | Pom |
| `spring-boot-starter-web-services` | Starter for using Spring Web Services | Pom |
| `spring-boot-starter-webflux` | Starter for building WebFlux applications using Spring Framework's Reactive Web support | Pom |
| `spring-boot-starter-websocket` | Starter for building WebSocket applications using Spring Framework's WebSocket support | Pom |

In addition to the application starters, the following starters can be used to add *production ready* features:

*Table 13.2. Spring Boot production starters*

| Name | Description | Pom |
|---|---|---|
| `spring-boot-starter-actuator` | Starter for using Spring Boot's Actuator which provides production ready features to help you monitor and manage your application | Pom |

Finally, Spring Boot also includes the following starters that can be used if you want to exclude or swap specific technical facets:

*Table 13.3. Spring Boot technical starters*

| Name | Description | Pom |
|---|---|---|
| `spring-boot-starter-jetty` | Starter for using Jetty as the embedded servlet container. An alternative to `spring-boot-starter-tomcat` | Pom |
| `spring-boot-starter-log4j2` | Starter for using Log4j2 for logging. An alternative to `spring-boot-starter-logging` | Pom |
| `spring-boot-starter-logging` | Starter for logging using Logback. Default logging starter | Pom |
| `spring-boot-starter-reactor-netty` | Starter for using Reactor Netty as the embedded reactive HTTP server. | Pom |
| `spring-boot-starter-tomcat` | Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by `spring-boot-starter-web` | Pom |
| `spring-boot-starter-undertow` | Starter for using Undertow as the embedded servlet container. An alternative to `spring-boot-starter-tomcat` | Pom |

**Tip**

For a list of additional community contributed starters, see the README file in the `spring-boot-starters` module on GitHub.

# 14. Structuring Your Code

Spring Boot does not require any specific code layout to work. However, there are some best practices that help.

## 14.1 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`, `@EntityScan`, or `@SpringBootApplication` annotations, since every class from every jar is read.

> **Tip**
>
> We recommend that you follow Java’s recommended package naming conventions and use a reversed domain name (for example, `com.example.project`).

## 14.2 Locating the Main Application Class

We generally recommend that you locate your main application class in a root package above other classes. The `@SpringBootApplication` annotation is often placed on your main class, and it implicitly defines a base “search package” for certain items. For example, if you are writing a JPA application, the package of the `@SpringBootApplication` annotated class is used to search for `@Entity` items. Using a root package also allows component scan to apply only on your project.

> **Tip**
>
> If you don’t want to use `@SpringBootApplication`, the `@EnableAutoConfiguration` and `@ComponentScan` annotations that it imports defines that behaviour so you can also use that instead.

The following listing shows a typical layout:

```
com
 +- example
     +- myapplication
         +- Application.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

The `Application.java` file would declare the `main` method, along with the basic `@SpringBootApplication`, as follows:

```
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
```

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

@SpringBootApplication
public class Application {

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

}
```

# 15. Configuration Classes

Spring Boot favors Java-based configuration. Although it is possible to use `SpringApplication` with XML sources, we generally recommend that your primary source be a single `@Configuration` class. Usually the class that defines the `main` method is a good candidate as the primary `@Configuration`.

> **Tip**
>
> Many Spring configuration examples have been published on the Internet that use XML configuration. If possible, always try to use the equivalent Java-based configuration. Searching for `Enable*` annotations can be a good starting point.

## 15.1 Importing Additional Configuration Classes

You need not put all your `@Configuration` into a single class. The `@Import` annotation can be used to import additional configuration classes. Alternatively, you can use `@ComponentScan` to automatically pick up all Spring components, including `@Configuration` classes.

## 15.2 Importing XML Configuration

If you absolutely must use XML based configuration, we recommend that you still start with a `@Configuration` class. You can then use an `@ImportResource` annotation to load XML configuration files.

# 16. Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, if `HSQLDB` is on your classpath, and you have not manually configured any database connection beans, then Spring Boot auto-configures an in-memory database.

You need to opt-in to auto-configuration by adding the `@EnableAutoConfiguration` or `@SpringBootApplication` annotations to one of your `@Configuration` classes.

> **Tip**
>
> You should only ever add one `@SpringBootApplication` or `@EnableAutoConfiguration` annotation. We generally recommend that you add one or the other to your primary `@Configuration` class only.

## 16.1 Gradually Replacing Auto-configuration

Auto-configuration is non-invasive. At any point, you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own `DataSource` bean, the default embedded database support backs away.

If you need to find out what auto-configuration is currently being applied, and why, start your application with the `--debug` switch. Doing so enables debug logs for a selection of core loggers and logs a conditions report to the console.

## 16.2 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 `@EnableAutoConfiguration` to disable them, as shown in the following example:

```
import org.springframework.boot.autoconfigure.*;
import org.springframework.boot.autoconfigure.jdbc.*;
import org.springframework.context.annotation.*;

@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

If the class is not on the classpath, you can use the `excludeName` attribute of the annotation and specify the fully qualified name instead. Finally, you can also control the list of auto-configuration classes to exclude by using the `spring.autoconfigure.exclude` property.

> **Tip**
>
> You can define exclusions both at the annotation level and by using the property.

# 17. Spring Beans and Dependency Injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. For simplicity, we often find that using `@ComponentScan` (to find your beans) and using `@Autowired` (to do constructor injection) works well.

If you structure your code as suggested above (locating your application class in a root package), you can add `@ComponentScan` without any arguments. All of your application components (`@Component`, `@Service`, `@Repository`, `@Controller` etc.) are automatically registered as Spring Beans.

The following example shows a `@Service` Bean that uses constructor injection to obtain a required `RiskAssessor` bean:

```
package com.example.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class DatabaseAccountService implements AccountService {

 private final RiskAssessor riskAssessor;

 @Autowired
 public DatabaseAccountService(RiskAssessor riskAssessor) {
  this.riskAssessor = riskAssessor;
 }

 // ...

}
```

If a bean has one constructor, you can omit the `@Autowired`, as shown in the following example:

```
@Service
public class DatabaseAccountService implements AccountService {

 private final RiskAssessor riskAssessor;

 public DatabaseAccountService(RiskAssessor riskAssessor) {
  this.riskAssessor = riskAssessor;
 }

 // ...

}
```

> **Tip**
>
> Notice how using constructor injection lets the `riskAssessor` field be marked as `final`, indicating that it cannot be subsequently changed.

# 18. Using the @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
- `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)
- `@Configuration`: allow to register extra beans in the context or import additional configuration classes

The `@SpringBootApplication` annotation is equivalent to using `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` with their default attributes, as shown in the following example:

```
package com.example.myapplication;

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

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
public class Application {

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

}
```

**Note**

`@SpringBootApplication` also provides aliases to customize the attributes of `@EnableAutoConfiguration` and `@ComponentScan`.

**Note**

None of these features are mandatory and you may chose to replace this single annotation by any of the features that it enables. For instance, you may not want to use component scan in your application:

```
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@EnableAutoConfiguration
@Import({ MyConfig.class, MyAnotherConfig.class })
public class Application {

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

}
```

> In this example, `Application` is just like any other Spring Boot application except that `@Component`-annotated classes are not detected automatically and the user-defined beans are imported explicitly (see `@Import`).

# 19. Running Your Application

One of the biggest advantages of packaging your application as a jar and using an embedded HTTP server is that you can run your application as you would any other. Debugging Spring Boot applications is also easy. You do not need any special IDE plugins or extensions.

> **Note**
>
> This section only covers jar based packaging. If you choose to package your application as a war file, you should refer to your server and IDE documentation.

## 19.1 Running from an IDE

You can run a Spring Boot application from your IDE as a simple Java application. However, you first need to import your project. Import steps vary depending on your IDE and build system. Most IDEs can import Maven projects directly. For example, Eclipse users can select `Import…` → `Existing Maven Projects` from the `File` menu.

If you cannot directly import your project into your IDE, you may be able to generate IDE metadata by using a build plugin. Maven includes plugins for Eclipse and IDEA. Gradle offers plugins for various IDEs.

> **Tip**
>
> If you accidentally run a web application twice, you see a “Port already in use” error. STS users can use the `Relaunch` button rather than the `Run` button to ensure that any existing instance is closed.

## 19.2 Running as a Packaged Application

If you use the Spring Boot Maven or Gradle plugins to create an executable jar, you can run your application using `java -jar`, as shown in the following example:

```
$ java -jar target/myapplication-0.0.1-SNAPSHOT.jar
```

It is also possible to run a packaged application with remote debugging support enabled. Doing so lets you attach a debugger to your packaged application, as shown in the following example:

```
$ java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000,suspend=n \
       -jar target/myapplication-0.0.1-SNAPSHOT.jar
```

## 19.3 Using the Maven Plugin

The Spring Boot Maven plugin includes a `run` goal that can be used to quickly compile and run your application. Applications run in an exploded form, as they do in your IDE. The following example shows a typical Maven command to run a Spring Boot application:

```
$ mvn spring-boot:run
```

You might also want to use the `MAVEN_OPTS` operating system environment variable, as shown in the following example:

```
$ export MAVEN_OPTS=-Xmx1024m
```

## 19.4 Using the Gradle Plugin

The Spring Boot Gradle plugin also includes a `bootRun` task that can be used to run your application in an exploded form. The `bootRun` task is added whenever you apply the `org.springframework.boot` and `java` plugins and is shown in the following example:

```
$ gradle bootRun
```

You might also want to use the `JAVA_OPTS` operating system environment variable, as shown in the following example:

```
$ export JAVA_OPTS=-Xmx1024m
```

## 19.5 Hot Swapping

Since Spring Boot applications are just plain Java applications, JVM hot-swapping should work out of the box. JVM hot swapping is somewhat limited with the bytecode that it can replace. For a more complete solution, JRebel can be used.

The `spring-boot-devtools` module also includes support for quick application restarts. See the Chapter 20, *Developer Tools* section later in this chapter and the Hot swapping “How-to” for details.

# 20. Developer Tools

Spring Boot includes an additional set of tools that can make the application development experience a little more pleasant. The `spring-boot-devtools` module can be included in any project to provide additional development-time features. To include devtools support, add the module dependency to your build, as shown in the following listings for Maven and Gradle:

**Maven.**

```
<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
 </dependency>
</dependencies>
```

**Gradle.**

```
dependencies {
 compile("org.springframework.boot:spring-boot-devtools")
}
```

> **Note**
>
> Developer tools are automatically disabled when running a fully packaged application. If your application is launched from `java -jar` or if it is started from a special classloader, then it is considered a “production application”. Flagging the dependency as optional in Maven or using `compileOnly` in Gradle is a best practice that prevents devtools from being transitively applied to other modules that use your project.

> **Tip**
>
> Repackaged archives do not contain devtools by default. If you want to use a certain remote devtools feature, you need to disable the `excludeDevtools` build property to include it. The property is supported with both the Maven and Gradle plugins.

## 20.1 Property Defaults

Several of the libraries supported by Spring Boot use caches to improve performance. For example, template engines cache compiled templates to avoid repeatedly parsing template files. Also, Spring MVC can add HTTP caching headers to responses when serving static resources.

While caching is very beneficial in production, it can be counter-productive during development, preventing you from seeing the changes you just made in your application. For this reason, spring-boot-devtools disables the caching options by default.

Cache options are usually configured by settings in your `application.properties` file. For example, Thymeleaf offers the `spring.thymeleaf.cache` property. Rather than needing to set these properties manually, the `spring-boot-devtools` module automatically applies sensible development-time configuration.

**Tip**

For a complete list of the properties that are applied by the devtools, see DevToolsPropertyDefaultsPostProcessor.

## 20.2 Automatic Restart

Applications that use `spring-boot-devtools` automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE, as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a folder is monitored for changes. Note that certain resources, such as static assets and view templates, do not need to restart the application.

**Triggering a restart**

As DevTools monitors classpath resources, the only way to trigger a restart is to update the classpath. The way in which you cause the classpath to be updated depends on the IDE that you are using. In Eclipse, saving a modified file causes the classpath to be updated and triggers a restart. In IntelliJ IDEA, building the project (`Build -> Build Project`) has the same effect.

**Note**

As long as forking is enabled, you can also start your application by using the supported build plugins (Maven and Gradle), since DevTools needs an isolated application classloader to operate properly. By default, Gradle and Maven do that when they detect DevTools on the classpath.

**Tip**

Automatic restart works very well when used with LiveReload. See the LiveReload section for details. If you use JRebel, automatic restarts are disabled in favor of dynamic class reloading. Other devtools features (such as LiveReload and property overrides) can still be used.

**Note**

DevTools relies on the application context's shutdown hook to close it during a restart. It does not work correctly if you have disabled the shutdown hook (`SpringApplication.setRegisterShutdownHook(false)`).

**Note**

When deciding if an entry on the classpath should trigger a restart when it changes, DevTools automatically ignores projects named `spring-boot`, `spring-boot-devtools`, `spring-boot-autoconfigure`, `spring-boot-actuator`, and `spring-boot-starter`.

**Note**

DevTools needs to customize the `ResourceLoader` used by the `ApplicationContext`. If your application provides one already, it is going to be wrapped. Direct override of the `getResource` method on the `ApplicationContext` is not supported.

**Restart vs Reload**

The restart technology provided by Spring Boot works by using two classloaders. Classes that do not change (for example, those from third-party jars) are loaded into a *base* classloader. Classes that you are actively developing are loaded into a *restart* classloader. When the application is restarted, the *restart* classloader is thrown away and a new one is created. This approach means that application restarts are typically much faster than “cold starts”, since the *base* classloader is already available and populated.

If you find that restarts are not quick enough for your applications or you encounter classloading issues, you could consider reloading technologies such as JRebel from ZeroTurnaround. These work by rewriting classes as they are loaded to make them more amenable to reloading.

## Logging changes in condition evaluation

By default, each time your application restarts, a report showing the condition evaluation delta is logged. The report shows the changes to your application’s auto-configuration as you make changes such as adding or removing beans and setting configuration properties.

To disable the logging of the report, set the following property:

```
spring.devtools.restart.log-condition-evaluation-delta=false
```

## Excluding Resources

Certain resources do not necessarily need to trigger a restart when they are changed. For example, Thymeleaf templates can be edited in-place. By default, changing resources in `/META-INF/maven`, `/META-INF/resources`, `/resources`, `/static`, `/public`, or `/templates` does not trigger a restart but does trigger a live reload. If you want to customize these exclusions, you can use the `spring.devtools.restart.exclude` property. For example, to exclude only `/static` and `/public` you would set the following property:

```
spring.devtools.restart.exclude=static/**,public/**
```

**Tip**

If you want to keep those defaults and *add* additional exclusions, use the `spring.devtools.restart.additional-exclude` property instead.

## Watching Additional Paths

You may want your application to be restarted or reloaded when you make changes to files that are not on the classpath. To do so, use the `spring.devtools.restart.additional-paths` property to configure additional paths to watch for changes. You can use the `spring.devtools.restart.exclude` property described earlier to control whether changes beneath the additional paths trigger a full restart or a live reload.

## Disabling Restart

If you do not want to use the restart feature, you can disable it by using the `spring.devtools.restart.enabled` property. In most cases, you can set this property in your `application.properties` (doing so still initializes the restart classloader, but it does not watch for file changes).

If you need to *completely* disable restart support (for example, because it does not work with a specific library), you need to set the `spring.devtools.restart.enabled System` property to `false` before calling `SpringApplication.run(…)`, as shown in the following example:

```
public static void main(String[] args) {
 System.setProperty("spring.devtools.restart.enabled", "false");
 SpringApplication.run(MyApp.class, args);
}
```

## Using a Trigger File

If you work with an IDE that continuously compiles changed files, you might prefer to trigger restarts only at specific times. To do so, you can use a “trigger file”, which is a special file that must be modified when you want to actually trigger a restart check. Changing the file only triggers the check and the restart only occurs if Devtools has detected it has to do something. The trigger file can be updated manually or with an IDE plugin.

To use a trigger file, set the `spring.devtools.restart.trigger-file` property to the path of your trigger file.

> **Tip**
>
> You might want to set `spring.devtools.restart.trigger-file` as a global setting, so that all your projects behave in the same way.

## Customizing the Restart Classloader

As described earlier in the Restart vs Reload section, restart functionality is implemented by using two classloaders. For most applications, this approach works well. However, it can sometimes cause classloading issues.

By default, any open project in your IDE is loaded with the “restart” classloader, and any regular `.jar` file is loaded with the “base” classloader. If you work on a multi-module project, and not every module is imported into your IDE, you may need to customize things. To do so, you can create a `META-INF/spring-devtools.properties` file.

The `spring-devtools.properties` file can contain properties prefixed with `restart.exclude` and `restart.include`. The `include` elements are items that should be pulled up into the “restart” classloader, and the `exclude` elements are items that should be pushed down into the “base” classloader. The value of the property is a regex pattern that is applied to the classpath, as shown in the following example:

```
restart.exclude.companycommonlibs=/mycorp-common-[\\w-]+\.jar
restart.include.projectcommon=/mycorp-myproj-[\\w-]+\.jar
```

> **Note**
>
> All property keys must be unique. As long as a property starts with `restart.include.` or `restart.exclude.` it is considered.

> **Tip**
>
> All `META-INF/spring-devtools.properties` from the classpath are loaded. You can package files inside your project, or in the libraries that the project consumes.

### Known Limitations

Restart functionality does not work well with objects that are deserialized by using a standard `ObjectInputStream`. If you need to deserialize data, you may need to use Spring's `ConfigurableObjectInputStream` in combination with `Thread.currentThread().getContextClassLoader()`.

Unfortunately, several third-party libraries deserialize without considering the context classloader. If you find such a problem, you need to request a fix with the original authors.

## 20.3 LiveReload

The `spring-boot-devtools` module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. LiveReload browser extensions are freely available for Chrome, Firefox and Safari from livereload.com.

If you do not want to start the LiveReload server when your application runs, you can set the `spring.devtools.livereload.enabled` property to `false`.

> **Note**
>
> You can only run one LiveReload server at a time. Before starting your application, ensure that no other LiveReload servers are running. If you start multiple applications from your IDE, only the first has LiveReload support.

## 20.4 Global Settings

You can configure global devtools settings by adding a file named `.spring-boot-devtools.properties` to your `$HOME` folder (note that the filename starts with "."). Any properties added to this file apply to *all* Spring Boot applications on your machine that use devtools. For example, to configure restart to always use a trigger file, you would add the following property:

**~/.spring-boot-devtools.properties.**

```
spring.devtools.reload.trigger-file=.reloadtrigger
```

## 20.5 Remote Applications

The Spring Boot developer tools are not limited to local development. You can also use several features when running applications remotely. Remote support is opt-in. To enable it, you need to make sure that `devtools` is included in the repackaged archive, as shown in the following listing:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <configuration>
    <excludeDevtools>false</excludeDevtools>
   </configuration>
  </plugin>
 </plugins>
</build>
```

Then you need to set a `spring.devtools.remote.secret` property, as shown in the following example:

```
spring.devtools.remote.secret=mysecret
```

**Warning**

Enabling `spring-boot-devtools` on a remote application is a security risk. You should never enable support on a production deployment.

Remote devtools support is provided in two parts: a server-side endpoint that accepts connections and a client application that you run in your IDE. The server component is automatically enabled when the `spring.devtools.remote.secret` property is set. The client component must be launched manually.

## Running the Remote Client Application

The remote client application is designed to be run from within your IDE. You need to run `org.springframework.boot.devtools.RemoteSpringApplication` with the same classpath as the remote project that you connect to. The application's single required argument is the remote URL to which it connects.

For example, if you are using Eclipse or STS and you have a project named `my-app` that you have deployed to Cloud Foundry, you would do the following:

- Select `Run Configurations...` from the `Run` menu.
- Create a new `Java Application` "launch configuration".
- Browse for the `my-app` project.
- Use `org.springframework.boot.devtools.RemoteSpringApplication` as the main class.
- Add `https://myapp.cfapps.io` to the `Program arguments` (or whatever your remote URL is).

A running remote client might resemble the following listing:

```
  .   ____          _                                              __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _          ___                     _      \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` |        | _ \___ _ __  ___| |_ ___ \ \ \ \
 \\/  ___)| |_)| | | | | || (_| []::::::[]   / -_) '  \/ _ \  _/ -_) ) ) ) )
  '  |____| .__|_| |_|_| |_\__, |        |_|_\___|_|_|_\___/\__\___|/ / / /
 =========|_|==============|___/=================================/_/_/_/
 :: Spring Boot Remote :: 2.1.0.BUILD-SNAPSHOT

2015-06-10 18:25:06.632  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringApplication   :
 Starting RemoteSpringApplication on pwmbp with PID 14938 (/Users/pwebb/projects/spring-boot/code/
spring-boot-devtools/target/classes started by pwebb in /Users/pwebb/projects/spring-boot/code/spring-
boot-samples/spring-boot-sample-devtools)
2015-06-10 18:25:06.671  INFO 14938 --- [           main] s.c.a.AnnotationConfigApplicationContext :
 Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@2a17b7b6: startup
 date [Wed Jun 10 18:25:06 PDT 2015]; root of context hierarchy
2015-06-10 18:25:07.043  WARN 14938 --- [           main] o.s.b.d.r.c.RemoteClientConfiguration    : The
 connection to http://localhost:8080 is insecure. You should use a URL starting with 'https://'.
2015-06-10 18:25:07.074  INFO 14938 --- [           main] o.s.b.d.a.OptionalLiveReloadServer       :
 LiveReload server is running on port 35729
2015-06-10 18:25:07.130  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringApplication   :
 Started RemoteSpringApplication in 0.74 seconds (JVM running for 1.105)
```

> **Note**
>
> Because the remote client is using the same classpath as the real application it can directly read application properties. This is how the `spring.devtools.remote.secret` property is read and passed to the server for authentication.

> **Tip**
>
> It is always advisable to use `https://` as the connection protocol, so that traffic is encrypted and passwords cannot be intercepted.

> **Tip**
>
> If you need to use a proxy to access the remote application, configure the `spring.devtools.remote.proxy.host` and `spring.devtools.remote.proxy.port` properties.

## Remote Update

The remote client monitors your application classpath for changes in the same way as the local restart. Any updated resource is pushed to the remote application and (*if required*) triggers a restart. This can be helpful if you iterate on a feature that uses a cloud service that you do not have locally. Generally, remote updates and restarts are much quicker than a full rebuild and deploy cycle.

> **Note**
>
> Files are only monitored when the remote client is running. If you change a file before starting the remote client, it is not pushed to the remote server.

# 21. Packaging Your Application for Production

Executable jars can be used for production deployment. As they are self-contained, they are also ideally suited for cloud-based deployment.

For additional “production ready” features, such as health, auditing, and metric REST or JMX end-points, consider adding `spring-boot-actuator`. See *Part V, “Spring Boot Actuator: Production-ready features”* for details.

# 22. What to Read Next

You should now understand how you can use Spring Boot and some best practices that you should follow. You can now go on to learn about specific *Spring Boot features* in depth, or you could skip ahead and read about the "production ready" aspects of Spring Boot.