Name	Description
spring-boot-starter-data-ldap	Starter for using Spring Data LDAP
spring-boot-starter-data-mongodb	Starter for using MongoDB document-oriented database and Spring Data MongoDB
spring-boot-starter-data-mongodb-reactive	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive
spring-boot-starter-data-neo4j	Starter for using Neo4j graph database and Spring Data Neo4j
spring-boot-starter-data-r2dbc	Starter for using Spring Data R2DBC
spring-boot-starter-data-redis	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client
spring-boot-starter-data-redis-reactive	Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client
spring-boot-starter-data-rest	Starter for exposing Spring Data repositories over REST using Spring Data REST and Spring MVC
spring-boot-starter-freemarker	Starter for building MVC web applications using FreeMarker views
spring-boot-starter-graphql	Starter for building GraphQL applications with Spring GraphQL
spring-boot-starter-groovy-templates	Starter for building MVC web applications using Groovy Templates views
spring-boot-starter-hateoas	Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS
spring-boot-starter-integration	Starter for using Spring Integration
spring-boot-starter-jdbc	Starter for using JDBC with the HikariCP connection pool
spring-boot-starter-jersey	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web
spring-boot-starter-jooq	Starter for using jOOQ to access SQL databases with JDBC. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc
spring-boot-starter-json	Starter for reading and writing json
spring-boot-starter-mail	Starter for using Java Mail and Spring Framework's email sending support
spring-boot-starter-mustache	Starter for building web applications using Mustache views

Name	Description
spring-boot-starter-oauth2-authorization-server	Starter for using Spring Authorization Server features
spring-boot-starter-oauth2-client	Starter for using Spring Security's OAuth2/OpenID Connect client features
spring-boot-starter-oauth2-resource-server	Starter for using Spring Security's OAuth2 resource server features
spring-boot-starter-pulsar	Starter for using Spring for Apache Pulsar
spring-boot-starter-pulsar-reactive	Starter for using Spring for Apache Pulsar Reactive
spring-boot-starter-quartz	Starter for using the Quartz scheduler
spring-boot-starter-rsocket	Starter for building RSocket clients and servers
spring-boot-starter-security	Starter for using Spring Security
spring-boot-starter-test	Starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito
spring-boot-starter-thymeleaf	Starter for building MVC web applications using Thymeleaf views
spring-boot-starter-validation	Starter for using Java Bean Validation with Hibernate Validator
spring-boot-starter-web	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
spring-boot-starter-web-services	Starter for using Spring Web Services
spring-boot-starter-webflux	Starter for building WebFlux applications using Spring Framework's Reactive Web support
spring-boot-starter-websocket	Starter for building WebSocket applications using Spring Framework's MVC WebSocket support

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

Table 2. Spring Boot production starters

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

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

Table 3. Spring Boot technical starters

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

To learn how to swap technical facets, please see the how-to documentation for swapping web server and logging system.

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

6.2. Structuring Your Code

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

TIP If you wish to enforce a structure based on domains, take a look at Spring Modulith.

6.2.1. 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`, `@ConfigurationPropertiesScan`, `@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`).

6.2.2. 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 do not want to use `@SpringBootApplication`, the `@EnableAutoConfiguration` and `@ComponentScan` annotations that it imports defines that behavior so you can also use those instead.

The following listing shows a typical layout:

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

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

Java

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

@SpringBootApplication
public class MyApplication {

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

}
```

Kotlin

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

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

6.3. 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.

6.3.1. 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.

6.3.2. 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.

6.4. 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.

6.4.1. 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.

6.4.2. 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 `@SpringBootApplication` to disable them, as shown in the following example:

Java

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

@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
public class MyApplication {

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

@SpringBootApplication(exclude = [DataSourceAutoConfiguration::class])
class MyApplication
```

If the class is not on the classpath, you can use the `excludeName` attribute of the annotation and specify the fully qualified name instead. If you prefer to use `@EnableAutoConfiguration` rather than `@SpringBootApplication`, `exclude` and `excludeName` are also available. 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.
NOTE	Even though auto-configuration classes are `public`, the only aspect of the class that is considered public API is the name of the class which can be used for disabling the auto-configuration. The actual contents of those classes, such as nested configuration classes or bean methods are for internal use only and we do not recommend using those directly.

6.4.3. Auto-configuration Packages

Auto-configuration packages are the packages that various auto-configured features look in by default when scanning for things such as entities and Spring Data repositories. The `@EnableAutoConfiguration` annotation (either directly or through its presence on `@SpringBootApplication`) determines the default auto-configuration package. Additional packages can be configured using the `@AutoConfigurationPackage` annotation.

6.5. 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. We 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.

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

Java

```
import org.springframework.stereotype.Service;

@Service
public class MyAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...

}
```

Kotlin

```
import org.springframework.stereotype.Service

@Service
class MyAccountService(private val riskAssessor: RiskAssessor) : AccountService
```

If a bean has more than one constructor, you will need to mark the one you want Spring to use with `@Autowired`:

Java

```
import java.io.PrintStream;

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

@Service
public class MyAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    private final PrintStream out;

    @Autowired
    public MyAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
        this.out = System.out;
    }

    public MyAccountService(RiskAssessor riskAssessor, PrintStream out) {
        this.riskAssessor = riskAssessor;
        this.out = out;
    }

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service
import java.io.PrintStream

@Service
class MyAccountService : AccountService {

    private val riskAssessor: RiskAssessor

    private val out: PrintStream

    @Autowired
    constructor(riskAssessor: RiskAssessor) {
        this.riskAssessor = riskAssessor
        out = System.out
    }

    constructor(riskAssessor: RiskAssessor, out: PrintStream) {
        this.riskAssessor = riskAssessor
        this.out = out
    }

    // ...

}
```

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

6.6. 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)
- `@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 in your integration tests.

Java

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

// Same as @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan
@SpringBootApplication
public class MyApplication {

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

}
```

Kotlin

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

// same as @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan
@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

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

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

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Import;

@SpringBootConfiguration(proxyBeanMethods = false)
@EnableAutoConfiguration
@Import({ SomeConfiguration.class, AnotherConfiguration.class })
public class MyApplication {

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

}
```

NOTE

Kotlin

```
import org.springframework.boot.SpringBootConfiguration
import org.springframework.boot.autoconfigure.EnableAutoConfiguration
import
org.springframework.boot.docs.using.structuringyourcode.locatingthemainc
lass.MyApplication
import org.springframework.boot.runApplication
import org.springframework.context.annotation.Import

@SpringBootConfiguration(proxyBeanMethods = false)
@EnableAutoConfiguration
@Import(SomeConfiguration::class, AnotherConfiguration::class)
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

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

6.7. 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. The sample applies to debugging Spring Boot applications. 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, see your server and IDE documentation.

6.7.1. Running From an IDE

You can run a Spring Boot application from your IDE as a 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. Spring Tools users can use the `Relaunch` button rather than the `Run` button to ensure that any existing instance is closed.

6.7.2. 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 -agentlib:jdwp=server=y,transport=dt_socket,address=8000,suspend=n \
       -jar target/myapplication-0.0.1-SNAPSHOT.jar
```

6.7.3. 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
```

6.7.4. 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
```

6.7.5. Hot Swapping

Since Spring Boot applications are 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 Hot swapping "How-to" for details.

6.8. 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 {
    developmentOnly("org.springframework.boot:spring-boot-devtools")
}
```

CAUTION	Devtools might cause classloading issues, in particular in multi-module projects. Diagnosing Classloading Issues explains how to diagnose and solve them.
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". You can control this behavior by using the `spring.devtools.restart.enabled` system property. To enable devtools, irrespective of the classloader used to launch your application, set the `-Dspring.devtools.restart.enabled=true` system property. This must not be done in a production environment where running devtools is a security risk. To disable devtools, exclude the dependency or set the `-Dspring.devtools.restart.enabled=false` system property.
TIP	Flagging the dependency as optional in Maven or using the `developmentOnly` configuration in Gradle (as shown above) 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 include it. When using the Maven plugin, set the `excludeDevtools` property to `false`. When using the Gradle plugin, configure the task's classpath to include the `developmentOnly` configuration.

6.8.1. Diagnosing Classloading Issues

As described 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, in particular in multi-module projects.

To diagnose whether the classloading issues are indeed caused by devtools and its two classloaders, try disabling restart. If this solves your problems, customize the restart classloader to include your entire project.

6.8.2. 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.

The following table lists all the properties that are applied:

Name	Default Value
`server.error.include-binding-errors`	`always`
`server.error.include-message`	`always`
`server.error.include-stacktrace`	`always`
`server.servlet.jsp.init-parameters.development`	`true`
`server.servlet.session.persistent`	`true`
`spring.docker.compose.readiness.wait`	`only-if-started`
`spring.freemarker.cache`	`false`
`spring.graphql.graphiql.enabled`	`true`
`spring.groovy.template.cache`	`false`
`spring.h2.console.enabled`	`true`
`spring.mustache.servlet.cache`	`false`
`spring.mvc.log-resolved-exception`	`true`
`spring.reactor.netty.shutdown-quiet-period`	`0s`
`spring.template.provider.cache`	`false`
`spring.thymeleaf.cache`	`false`
`spring.web.resources.cache.period`	`0`
`spring.web.resources.chain.cache`	`false`

NOTE If you do not want property defaults to be applied you can set `spring.devtools.add-properties` to `false` in your `application.properties`.

Because you need more information about web requests while developing Spring MVC and Spring WebFlux applications, developer tools suggests you to enable `DEBUG` logging for the `web` logging group. This will give you information about the incoming request, which handler is processing it, the response outcome, and other details. If you wish to log all request details (including potentially sensitive information), you can turn on the `spring.mvc.log-request-details` or `spring.codec.log-request-details` configuration properties.

6.8.3. 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 directory 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. Whether you're using an IDE or one of the build plugins, the modified files have to be recompiled to trigger a restart. The way in which you cause the classpath to be updated depends on the tool 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.
- If using a build plugin, running `mvn compile` for Maven or `gradle build` for Gradle will trigger a restart.

NOTE If you are restarting with Maven or Gradle using the build plugin you must leave the `forking` set to `enabled`. If you disable forking, the isolated application classloader used by devtools will not be created and restarts will not operate properly.

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 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.

CAUTION Automatic restart is not supported when using AspectJ weaving.

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:

Properties

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

Yaml

```
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:

Properties

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

Yaml

```
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:

Java

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

@SpringBootApplication
public class MyApplication {

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

}
```

Kotlin

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

@SpringBootApplication
object MyApplication {

    @JvmStatic
    fun main(args: Array<String>) {
        System.setProperty("spring.devtools.restart.enabled", "false")
        SpringApplication.run(MyApplication::class.java, *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.

NOTE	Any update to the file will trigger a check, but restart only actually occurs if Devtools has detected it has something to do.

To use a trigger file, set the `spring.devtools.restart.trigger-file` property to the name (excluding any path) of your trigger file. The trigger file must appear somewhere on your classpath.

For example, if you have a project with the following structure:

```
src
+- main
   +- resources
      +- .reloadtrigger
```

Then your `trigger-file` property would be:

Properties

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

Yaml

```
spring:
  devtools:
    restart:
      trigger-file: ".reloadtrigger"
```

Restarts will now only happen when the `src/main/resources/.reloadtrigger` is updated.

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.

Some IDEs have features that save you from needing to update your trigger file manually. Spring Tools for Eclipse and IntelliJ IDEA (Ultimate Edition) both have such support. With Spring Tools, you can use the "reload" button from the console view (as long as your `trigger-file` is named `.reloadtrigger`). For IntelliJ IDEA, you can follow the instructions in their documentation.

Customizing the Restart Classloader

As described earlier in the Restart vs Reload section, restart functionality is implemented by using two classloaders. If this causes issues, you can diagnose the problem by using the `spring.devtools.restart.enabled` system property, and if the app works with restart switched off, you might need to customize what gets loaded by which classloader.

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. The same is true if you use `mvn spring-boot:run` or `gradle bootRun`: the project containing your `@SpringBootApplication` is loaded with the "restart" classloader, and everything else with the "base" classloader. The classpath is printed on the console when you start the app, which can help to identify any problematic entries. Classes used reflectively, especially annotations, can be loaded into the parent (fixed) classloader on startup before the application classes which uses them, and this might lead to them not being detected by Spring in the application.

You can instruct Spring Boot to load parts of your project with a different classloader by creating 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 passed to the JVM on startup. Here is an example where some local class files are excluded and some extra libraries are included in the restart class loader:

Properties

```
restart.exclude.companycommonlibs=/mycorp-common-[\\w\\d-\\.]/(build|bin|out|target)/
restart.include.projectcommon=/mycorp-myproj-[\\w\\d-\\.]+\\.jar
```

Yaml

```
restart:
  exclude:
    companycommonlibs: "/mycorp-common-[\\w\\d-\\.]/(build|bin|out|target)/"
  include:
    projectcommon: "/mycorp-myproj-[\\w\\d-\\.]+\\.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. System properties can not be used, only the properties file.

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.

6.8.4. 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. You can find these extensions by searching 'LiveReload' in the marketplace or store of your chosen browser.

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.

WARNING	To trigger LiveReload when a file changes, Automatic Restart must be enabled.

6.8.5. Global Settings

You can configure global devtools settings by adding any of the following files to the `$HOME/.config/spring-boot` directory:

1. `spring-boot-devtools.properties`
2. `spring-boot-devtools.yaml`
3. `spring-boot-devtools.yml`

Any properties added to these files 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 to your `spring-boot-devtools` file:

Properties

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

Yaml

```
spring:
  devtools:
    restart:
      trigger-file: ".reloadtrigger"
```

By default, `$HOME` is the user's home directory. To customize this location, set the `SPRING_DEVTOOLS_HOME` environment variable or the `spring.devtools.home` system property.

NOTE	If devtools configuration files are not found in `$HOME/.config/spring-boot`, the root of the `$HOME` directory is searched for the presence of a `.spring-boot-devtools.properties` file. This allows you to share the devtools global configuration with applications that are on an older version of Spring Boot that does not support the `$HOME/.config/spring-boot` location.
NOTE	Profiles are not supported in devtools properties/yaml files. Any profiles activated in `.spring-boot-devtools.properties` will not affect the loading of profile-specific configuration files. Profile specific filenames (of the form `spring-boot-devtools-<profile>.properties`) and `spring.config.activate.on-profile` documents in both YAML and Properties files are not supported.

Configuring File System Watcher

FileSystemWatcher works by polling the class changes with a certain time interval, and then waiting for a predefined quiet period to make sure there are no more changes. Since Spring Boot relies entirely on the IDE to compile and copy files into the location from where Spring Boot can read them, you might find that there are times when certain changes are not reflected when devtools restarts the application. If you observe such problems constantly, try increasing the `spring.devtools.restart.poll-interval` and `spring.devtools.restart.quiet-period` parameters to the values that fit your development environment:

Properties

```
spring.devtools.restart.poll-interval=2s
spring.devtools.restart.quiet-period=1s
```

Yaml

```
spring:
  devtools:
    restart:
      poll-interval: "2s"
      quiet-period: "1s"
```

The monitored classpath directories are now polled every 2 seconds for changes, and a 1 second quiet period is maintained to make sure there are no additional class changes.

6.8.6. 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 as enabling it can be a security risk. It should only be enabled when running on a trusted network or when secured with SSL. If neither of these options is available to you, you should not use DevTools' remote support. You should never enable support on a production deployment.

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 the `spring.devtools.remote.secret` property. Like any important password or secret, the value should be unique and strong such that it cannot be guessed or brute-forced.

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.

NOTE	Remote devtools is not supported for Spring WebFlux applications.

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 Spring Tools 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 ::  (v3.2.7)

2024-06-20T08:04:29.477Z  INFO 112392 --- [           main]
o.s.b.devtools.RemoteSpringApplication   : Starting RemoteSpringApplication v3.2.7
using Java 17.0.11 with PID 112392
(/Users/myuser/.m2/repository/org/springframework/boot/spring-boot-
devtools/3.2.7/spring-boot-devtools-3.2.7.jar started by myuser in /opt/apps/)
2024-06-20T08:04:29.495Z  INFO 112392 --- [           main]
o.s.b.devtools.RemoteSpringApplication   : No active profile set, falling back to 1
default profile: "default"
2024-06-20T08:04:31.071Z  INFO 112392 --- [           main]
o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2024-06-20T08:04:31.171Z  INFO 112392 --- [           main]
o.s.b.devtools.RemoteSpringApplication   : Started RemoteSpringApplication in 2.929
seconds (process running for 4.401)
```

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.

On a slower development environment, it may happen that the quiet period is not enough, and the changes in the classes may be split into batches. The server is restarted after the first batch of class changes is uploaded. The next batch can't be sent to the application, since the server is restarting.

This is typically manifested by a warning in the `RemoteSpringApplication` logs about failing to upload some of the classes, and a consequent retry. But it may also lead to application code inconsistency and failure to restart after the first batch of changes is uploaded. If you observe such problems constantly, try increasing the `spring.devtools.restart.poll-interval` and `spring.devtools.restart.quiet-period` parameters to the values that fit your development environment. See the Configuring File System Watcher section for configuring these properties.

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.

6.9. 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 *Production-ready Features* for details.

6.10. 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.

Chapter 7. Core Features

This section dives into the details of Spring Boot. Here you can learn about the key features that you may want to use and customize. If you have not already done so, you might want to read the "Getting Started" and "Developing with Spring Boot" sections, so that you have a good grounding of the basics.

7.1. SpringApplication

The `SpringApplication` class provides a convenient way to bootstrap a Spring application that is started from a `main()` method. In many situations, you can delegate to the static `SpringApplication.run` method, as shown in the following example:

Java

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

@SpringBootApplication
public class MyApplication {

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

}
```

Kotlin

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


@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args)
}
```

When your application starts, you should see something similar to the following output:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.2.7)

2024-06-20T08:04:34.953Z  INFO 112644 --- [           main]
o.s.b.d.f.logexample.MyApplication       : Starting MyApplication using Java 17.0.11
with PID 112644 (/opt/apps/myapp.jar started by myuser in /opt/apps/)
2024-06-20T08:04:35.034Z  INFO 112644 --- [           main]
o.s.b.d.f.logexample.MyApplication       : No active profile set, falling back to 1
default profile: "default"
2024-06-20T08:04:39.936Z  INFO 112644 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2024-06-20T08:04:40.030Z  INFO 112644 --- [           main]
o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-06-20T08:04:40.039Z  INFO 112644 --- [           main]
o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache
Tomcat/10.1.25]
2024-06-20T08:04:40.281Z  INFO 112644 --- [           main]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded
WebApplicationContext
2024-06-20T08:04:40.308Z  INFO 112644 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 5028 ms
2024-06-20T08:04:42.197Z  INFO 112644 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with
context path ''
2024-06-20T08:04:42.219Z  INFO 112644 --- [           main]
o.s.b.d.f.logexample.MyApplication       : Started MyApplication in 9.396 seconds
(process running for 10.948)
```

By default, `INFO` logging messages are shown, including some relevant startup details, such as the user that launched the application. If you need a log level other than `INFO`, you can set it, as described in Log Levels. The application version is determined using the implementation version from the main application class's package. Startup information logging can be turned off by setting `spring.main.log-startup-info` to `false`. This will also turn off logging of the application's active profiles.

TIP To add additional logging during startup, you can override `logStartupInfo(boolean)` in a subclass of `SpringApplication`.

7.1.1. Startup Failure

If your application fails to start, registered `FailureAnalyzers` get a chance to provide a dedicated error message and a concrete action to fix the problem. For instance, if you start a web application

on port `8080` and that port is already in use, you should see something similar to the following message:

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Embedded servlet container failed to start. Port 8080 was already in use.

Action:

Identify and stop the process that is listening on port 8080 or configure this
application to listen on another port.
```

NOTE	Spring Boot provides numerous `FailureAnalyzer` implementations, and you can add your own.

If no failure analyzers are able to handle the exception, you can still display the full conditions report to better understand what went wrong. To do so, you need to enable the `debug` property or enable `DEBUG` logging for `org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener`.

For instance, if you are running your application by using `java -jar`, you can enable the `debug` property as follows:

```
$ java -jar myproject-0.0.1-SNAPSHOT.jar --debug
```

7.1.2. Lazy Initialization

`SpringApplication` allows an application to be initialized lazily. When lazy initialization is enabled, beans are created as they are needed rather than during application startup. As a result, enabling lazy initialization can reduce the time that it takes your application to start. In a web application, enabling lazy initialization will result in many web-related beans not being initialized until an HTTP request is received.

A downside of lazy initialization is that it can delay the discovery of a problem with the application. If a misconfigured bean is initialized lazily, a failure will no longer occur during startup and the problem will only become apparent when the bean is initialized. Care must also be taken to ensure that the JVM has sufficient memory to accommodate all of the application's beans and not just those that are initialized during startup. For these reasons, lazy initialization is not enabled by default and it is recommended that fine-tuning of the JVM's heap size is done before enabling lazy initialization.

Lazy initialization can be enabled programmatically using the `lazyInitialization` method on `SpringApplicationBuilder` or the `setLazyInitialization` method on `SpringApplication`. Alternatively,

it can be enabled using the `spring.main.lazy-initialization` property as shown in the following example:

Properties

```
spring.main.lazy-initialization=true
```

Yaml

```
spring:
  main:
    lazy-initialization: true
```

TIP If you want to disable lazy initialization for certain beans while using lazy initialization for the rest of the application, you can explicitly set their lazy attribute to false using the `@Lazy(false)` annotation.

7.1.3. Customizing the Banner

The banner that is printed on start up can be changed by adding a `banner.txt` file to your classpath or by setting the `spring.banner.location` property to the location of such a file. If the file has an encoding other than UTF-8, you can set `spring.banner.charset`.

Inside your `banner.txt` file, you can use any key available in the `Environment` as well as any of the following placeholders:

Table 4. Banner variables

Variable	Description
`${application.version}`	The version number of your application, as declared in `MANIFEST.MF`. For example, `Implementation-Version: 1.0` is printed as `1.0`.
`${application.formatted-version}`	The version number of your application, as declared in `MANIFEST.MF` and formatted for display (surrounded with brackets and prefixed with `v`). For example `(v1.0)`.
`${spring-boot.version}`	The Spring Boot version that you are using. For example `3.2.7`.
`${spring-boot.formatted-version}`	The Spring Boot version that you are using, formatted for display (surrounded with brackets and prefixed with `v`). For example `(v3.2.7)`.
`${Ansi.NAME}` (or `${AnsiColor.NAME}`, `${AnsiBackground.NAME}`, `${AnsiStyle.NAME}`)	Where `NAME` is the name of an ANSI escape code. See `AnsiPropertySource` for details.
`${application.title}`	The title of your application, as declared in `MANIFEST.MF`. For example `Implementation-Title: MyApp` is printed as `MyApp`.

TIP

The `SpringApplication.setBanner(…)` method can be used if you want to generate a banner programmatically. Use the `org.springframework.boot.Banner` interface and implement your own `printBanner()` method.

You can also use the `spring.main.banner-mode` property to determine if the banner has to be printed on `System.out` (`console`), sent to the configured logger (`log`), or not produced at all (`off`).

The printed banner is registered as a singleton bean under the following name: `springBootBanner`.

NOTE

The `application.title`, `application.version`, and `application.formatted-version` properties are only available if you are using `java -jar` or `java -cp` with Spring Boot launchers. The values will not be resolved if you are running an unpacked jar and starting it with `java -cp <classpath> <mainclass>` or running your application as a native image.

To use the `application.` properties, launch your application as a packed jar using `java -jar` or as an unpacked jar using `java org.springframework.boot.loader.launch.JarLauncher`. This will initialize the `application.` banner properties before building the classpath and launching your app.

7.1.4. Customizing SpringApplication

If the `SpringApplication` defaults are not to your taste, you can instead create a local instance and customize it. For example, to turn off the banner, you could write:

Java

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

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(MyApplication.class);
        application.setBannerMode(Banner.Mode.OFF);
        application.run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.Banner
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args) {
        setBannerMode(Banner.Mode.OFF)
    }
}
```

NOTE	The constructor arguments passed to `SpringApplication` are configuration sources for Spring beans. In most cases, these are references to `@Configuration` classes, but they could also be direct references `@Component` classes.

It is also possible to configure the `SpringApplication` by using an `application.properties` file. See *Externalized Configuration* for details.

For a complete list of the configuration options, see the `SpringApplication` Javadoc.

7.1.5. Fluent Builder API

If you need to build an `ApplicationContext` hierarchy (multiple contexts with a parent/child relationship) or if you prefer using a "fluent" builder API, you can use the `SpringApplicationBuilder`.

The `SpringApplicationBuilder` lets you chain together multiple method calls and includes `parent` and `child` methods that let you create a hierarchy, as shown in the following example:

Java

```
new SpringApplicationBuilder().sources(Parent.class)
    .child(Application.class)
    .bannerMode(Banner.Mode.OFF)
    .run(args);
```

Kotlin

```
SpringApplicationBuilder()
    .sources(Parent::class.java)
    .child(Application::class.java)
    .bannerMode(Banner.Mode.OFF)
    .run(*args)
```

NOTE	There are some restrictions when creating an `ApplicationContext` hierarchy. For example, Web components **must** be contained within the child context, and the same `Environment` is used for both parent and child contexts. See the `SpringApplicationBuilder` Javadoc for full details.

7.1.6. Application Availability

When deployed on platforms, applications can provide information about their availability to the platform using infrastructure such as Kubernetes Probes. Spring Boot includes out-of-the box support for the commonly used "liveness" and "readiness" availability states. If you are using Spring Boot's "actuator" support then these states are exposed as health endpoint groups.

In addition, you can also obtain availability states by injecting the `ApplicationAvailability` interface into your own beans.

Liveness State

The "Liveness" state of an application tells whether its internal state allows it to work correctly, or recover by itself if it is currently failing. A broken "Liveness" state means that the application is in a state that it cannot recover from, and the infrastructure should restart the application.

NOTE	In general, the "Liveness" state should not be based on external checks, such as Health checks. If it did, a failing external system (a database, a Web API, an external cache) would trigger massive restarts and cascading failures across the platform.

The internal state of Spring Boot applications is mostly represented by the Spring `ApplicationContext`. If the application context has started successfully, Spring Boot assumes that the application is in a valid state. An application is considered live as soon as the context has been refreshed, see Spring Boot application lifecycle and related Application Events.

Readiness State

The "Readiness" state of an application tells whether the application is ready to handle traffic. A failing "Readiness" state tells the platform that it should not route traffic to the application for now. This typically happens during startup, while `CommandLineRunner` and `ApplicationRunner` components are being processed, or at any time if the application decides that it is too busy for additional traffic.

An application is considered ready as soon as application and command-line runners have been called, see Spring Boot application lifecycle and related Application Events.

TIP	Tasks expected to run during startup should be executed by `CommandLineRunner` and `ApplicationRunner` components instead of using Spring component lifecycle callbacks such as `@PostConstruct`.

Managing the Application Availability State

Application components can retrieve the current availability state at any time, by injecting the `ApplicationAvailability` interface and calling methods on it. More often, applications will want to

listen to state updates or update the state of the application.

For example, we can export the "Readiness" state of the application to a file so that a Kubernetes "exec Probe" can look at this file:

Java

```
import org.springframework.boot.availability.AvailabilityChangeEvent;
import org.springframework.boot.availability.ReadinessState;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
public class MyReadinessStateExporter {

    @EventListener
    public void onStateChange(AvailabilityChangeEvent<ReadinessState> event) {
        switch (event.getState()) {
            case ACCEPTING_TRAFFIC -> {
                // create file /tmp/healthy
            }
            case REFUSING_TRAFFIC -> {
                // remove file /tmp/healthy
            }
        }
    }

}
```

Kotlin

```
import org.springframework.boot.availability.AvailabilityChangeEvent
import org.springframework.boot.availability.ReadinessState
import org.springframework.context.event.EventListener
import org.springframework.stereotype.Component

@Component
class MyReadinessStateExporter {

    @EventListener
    fun onStateChange(event: AvailabilityChangeEvent<ReadinessState?>) {
        when (event.state) {
            ReadinessState.ACCEPTING_TRAFFIC -> {
                // create file /tmp/healthy
            }
            ReadinessState.REFUSING_TRAFFIC -> {
                // remove file /tmp/healthy
            }
            else -> {
                // ...
            }
        }
    }

}
```

We can also update the state of the application, when the application breaks and cannot recover:

Java

```
import org.springframework.boot.availability.AvailabilityChangeEvent;
import org.springframework.boot.availability.LivenessState;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.stereotype.Component;

@Component
public class MyLocalCacheVerifier {

    private final ApplicationEventPublisher eventPublisher;

    public MyLocalCacheVerifier(ApplicationEventPublisher eventPublisher) {
        this.eventPublisher = eventPublisher;
    }

    public void checkLocalCache() {
        try {
            // ...
        }
        catch (CacheCompletelyBrokenException ex) {
            AvailabilityChangeEvent.publish(this.eventPublisher, ex,
LivenessState.BROKEN);
        }
    }

}
```

Kotlin

```
import org.springframework.boot.availability.AvailabilityChangeEvent
import org.springframework.boot.availability.LivenessState
import org.springframework.context.ApplicationEventPublisher
import org.springframework.stereotype.Component

@Component
class MyLocalCacheVerifier(private val eventPublisher: ApplicationEventPublisher) {

    fun checkLocalCache() {
        try {
            // ...
        } catch (ex: CacheCompletelyBrokenException) {
            AvailabilityChangeEvent.publish(eventPublisher, ex, LivenessState.BROKEN)
        }
    }

}
```

Spring Boot provides Kubernetes HTTP probes for "Liveness" and "Readiness" with Actuator Health Endpoints. You can get more guidance about deploying Spring Boot applications on Kubernetes in

the dedicated section.

7.1.7. Application Events and Listeners

In addition to the usual Spring Framework events, such as `ContextRefreshedEvent`, a `SpringApplication` sends some additional application events.

NOTE

Some events are actually triggered before the `ApplicationContext` is created, so you cannot register a listener on those as a `@Bean`. You can register them with the `SpringApplication.addListeners(…)` method or the `SpringApplicationBuilder.listeners(…)` method.

If you want those listeners to be registered automatically, regardless of the way the application is created, you can add a `META-INF/spring.factories` file to your project and reference your listener(s) by using the `org.springframework.context.ApplicationListener` key, as shown in the following example:

```
org.springframework.context.ApplicationListener=com.example.project.MyListener
```

Application events are sent in the following order, as your application runs:

1. An `ApplicationStartingEvent` is sent at the start of a run but before any processing, except for the registration of listeners and initializers.
2. An `ApplicationEnvironmentPreparedEvent` is sent when the `Environment` to be used in the context is known but before the context is created.
3. An `ApplicationContextInitializedEvent` is sent when the `ApplicationContext` is prepared and ApplicationContextInitializers have been called but before any bean definitions are loaded.
4. An `ApplicationPreparedEvent` is sent just before the refresh is started but after bean definitions have been loaded.
5. An `ApplicationStartedEvent` is sent after the context has been refreshed but before any application and command-line runners have been called.
6. An `AvailabilityChangeEvent` is sent right after with `LivenessState.CORRECT` to indicate that the application is considered as live.
7. An `ApplicationReadyEvent` is sent after any application and command-line runners have been called.
8. An `AvailabilityChangeEvent` is sent right after with `ReadinessState.ACCEPTING_TRAFFIC` to indicate that the application is ready to service requests.
9. An `ApplicationFailedEvent` is sent if there is an exception on startup.

The above list only includes `SpringApplicationEvent`s that are tied to a `SpringApplication`. In addition to these, the following events are also published after `ApplicationPreparedEvent` and before `ApplicationStartedEvent`:

- A `WebServerInitializedEvent` is sent after the `WebServer` is ready. `ServletWebServerInitializedEvent` and `ReactiveWebServerInitializedEvent` are the servlet and reactive variants respectively.
- A `ContextRefreshedEvent` is sent when an `ApplicationContext` is refreshed.

TIP	You often need not use application events, but it can be handy to know that they exist. Internally, Spring Boot uses events to handle a variety of tasks.

NOTE	Event listeners should not run potentially lengthy tasks as they execute in the same thread by default. Consider using application and command-line runners instead.

Application events are sent by using Spring Framework's event publishing mechanism. Part of this mechanism ensures that an event published to the listeners in a child context is also published to the listeners in any ancestor contexts. As a result of this, if your application uses a hierarchy of `SpringApplication` instances, a listener may receive multiple instances of the same type of application event.

To allow your listener to distinguish between an event for its context and an event for a descendant context, it should request that its application context is injected and then compare the injected context with the context of the event. The context can be injected by implementing `ApplicationContextAware` or, if the listener is a bean, by using `@Autowired`.

7.1.8. Web Environment

A `SpringApplication` attempts to create the right type of `ApplicationContext` on your behalf. The algorithm used to determine a `WebApplicationType` is the following:

- If Spring MVC is present, an `AnnotationConfigServletWebServerApplicationContext` is used
- If Spring MVC is not present and Spring WebFlux is present, an `AnnotationConfigReactiveWebServerApplicationContext` is used
- Otherwise, `AnnotationConfigApplicationContext` is used

This means that if you are using Spring MVC and the new `WebClient` from Spring WebFlux in the same application, Spring MVC will be used by default. You can override that easily by calling `setWebApplicationType(WebApplicationType)`.

It is also possible to take complete control of the `ApplicationContext` type that is used by calling `setApplicationContextFactory(…)`.

TIP	It is often desirable to call `setWebApplicationType(WebApplicationType.NONE)` when using `SpringApplication` within a JUnit test.

7.1.9. Accessing Application Arguments

If you need to access the application arguments that were passed to `SpringApplication.run(…)`, you can inject a `org.springframework.boot.ApplicationArguments` bean. The `ApplicationArguments` interface provides access to both the raw `String[]` arguments as well as parsed `option` and `non-`

`option` arguments, as shown in the following example:

Java

```
import java.util.List;

import org.springframework.boot.ApplicationArguments;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    public MyBean(ApplicationArguments args) {
        boolean debug = args.containsOption("debug");
        List<String> files = args.getNonOptionArgs();
        if (debug) {
            System.out.println(files);
        }
        // if run with "--debug logfile.txt" prints ["logfile.txt"]
    }

}
```

Kotlin

```
import org.springframework.boot.ApplicationArguments
import org.springframework.stereotype.Component

@Component
class MyBean(args: ApplicationArguments) {

    init {
        val debug = args.containsOption("debug")
        val files = args.nonOptionArgs
        if (debug) {
            println(files)
        }
        // if run with "--debug logfile.txt" prints ["logfile.txt"]
    }

}
```

TIP Spring Boot also registers a `CommandLinePropertySource` with the Spring `Environment`. This lets you also inject single application arguments by using the `@Value` annotation.

7.1.10. Using the ApplicationRunner or CommandLineRunner

If you need to run some specific code once the `SpringApplication` has started, you can implement the `ApplicationRunner` or `CommandLineRunner` interfaces. Both interfaces work in the same way and

offer a single `run` method, which is called just before `SpringApplication.run(…)` completes.

NOTE	This contract is well suited for tasks that should run after application startup but before it starts accepting traffic.

The `CommandLineRunner` interfaces provides access to application arguments as a string array, whereas the `ApplicationRunner` uses the `ApplicationArguments` interface discussed earlier. The following example shows a `CommandLineRunner` with a `run` method:

Java

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

@Component
public class MyCommandLineRunner implements CommandLineRunner {

    @Override
    public void run(String... args) {
        // Do something...
    }

}
```

Kotlin

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

@Component
class MyCommandLineRunner : CommandLineRunner {

    override fun run(vararg args: String) {
        // Do something...
    }

}
```

If several `CommandLineRunner` or `ApplicationRunner` beans are defined that must be called in a specific order, you can additionally implement the `org.springframework.core.Ordered` interface or use the `org.springframework.core.annotation.Order` annotation.

7.1.11. Application Exit

Each `SpringApplication` registers a shutdown hook with the JVM to ensure that the `ApplicationContext` closes gracefully on exit. All the standard Spring lifecycle callbacks (such as the `DisposableBean` interface or the `@PreDestroy` annotation) can be used.

In addition, beans may implement the `org.springframework.boot.ExitCodeGenerator` interface if they

wish to return a specific exit code when `SpringApplication.exit()` is called. This exit code can then be passed to `System.exit()` to return it as a status code, as shown in the following example:

Java

```
import org.springframework.boot.ExitCodeGenerator;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class MyApplication {

    @Bean
    public ExitCodeGenerator exitCodeGenerator() {
        return () -> 42;
    }

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

}
```

Kotlin

```
import org.springframework.boot.ExitCodeGenerator
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.context.annotation.Bean

import kotlin.system.exitProcess

@SpringBootApplication
class MyApplication {

    @Bean
    fun exitCodeGenerator() = ExitCodeGenerator { 42 }

}

fun main(args: Array<String>) {
    exitProcess(SpringApplication.exit(
        runApplication<MyApplication>(*args)))
}
```

Also, the `ExitCodeGenerator` interface may be implemented by exceptions. When such an exception is encountered, Spring Boot returns the exit code provided by the implemented `getExitCode()`

method.

If there is more than one `ExitCodeGenerator`, the first non-zero exit code that is generated is used. To control the order in which the generators are called, additionally implement the `org.springframework.core.Ordered` interface or use the `org.springframework.core.annotation.Order` annotation.

7.1.12. Admin Features

It is possible to enable admin-related features for the application by specifying the `spring.application.admin.enabled` property. This exposes the `SpringApplicationAdminMXBean` on the platform `MBeanServer`. You could use this feature to administer your Spring Boot application remotely. This feature could also be useful for any service wrapper implementation.

TIP	If you want to know on which HTTP port the application is running, get the property with a key of `local.server.port`.

7.1.13. Application Startup tracking

During the application startup, the `SpringApplication` and the `ApplicationContext` perform many tasks related to the application lifecycle, the beans lifecycle or even processing application events. With `ApplicationStartup`, Spring Framework allows you to track the application startup sequence with `StartupStep` objects. This data can be collected for profiling purposes, or just to have a better understanding of an application startup process.

You can choose an `ApplicationStartup` implementation when setting up the `SpringApplication` instance. For example, to use the `BufferingApplicationStartup`, you could write:

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.metrics.buffering.BufferingApplicationStartup;

@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(MyApplication.class);
        application.setApplicationStartup(new BufferingApplicationStartup(2048));
        application.run(args);
    }

}
```

Kotlin

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.context.metrics.buffering.BufferingApplicationStartup
import org.springframework.boot.runApplication

@SpringBootApplication
class MyApplication

fun main(args: Array<String>) {
    runApplication<MyApplication>(*args) {
        applicationStartup = BufferingApplicationStartup(2048)
    }
}
```

The first available implementation, `FlightRecorderApplicationStartup` is provided by Spring Framework. It adds Spring-specific startup events to a Java Flight Recorder session and is meant for profiling applications and correlating their Spring context lifecycle with JVM events (such as allocations, GCs, class loading…). Once configured, you can record data by running the application with the Flight Recorder enabled:

```
$ java -XX:StartFlightRecording:filename=recording.jfr,duration=10s -jar demo.jar
```

Spring Boot ships with the `BufferingApplicationStartup` variant; this implementation is meant for buffering the startup steps and draining them into an external metrics system. Applications can ask for the bean of type `BufferingApplicationStartup` in any component.

Spring Boot can also be configured to expose a `startup` endpoint that provides this information as a JSON document.

7.1.14. Virtual threads

If you're running on Java 21 or up, you can enable virtual threads by setting the property `spring.threads.virtual.enabled` to `true`.

Before turning on this option for your application, you should consider reading the official Java virtual threads documentation. In some cases, applications can experience lower throughput because of "Pinned Virtual Threads"; this page also explains how to detect such cases with JDK Flight Recorder or the `jcmd` CLI.

WARNING	One side effect of virtual threads is that they are daemon threads. A JVM will exit if all of its threads are daemon threads. This behavior can be a problem when you rely on `@Scheduled` beans, for example, to keep your application alive. If you use virtual threads, the scheduler thread is a virtual thread and therefore a daemon thread and won't keep the JVM alive. This not only affects scheduling and can be the case with other technologies too. To keep the JVM running in all cases, it is recommended to set the property `spring.main.keep-alive` to `true`. This ensures that the JVM is kept alive, even if all threads are virtual threads.

7.2. 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` 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` and the test annotations for testing a particular slice of your application.
13. `@DynamicPropertySource` annotations in your tests.
14. `@TestPropertySource` annotations on your tests.

15. Devtools global settings properties in the `$HOME/.config/spring-boot` directory when devtools is active.

Config data files are considered in the following order:

1. Application properties packaged inside your jar (`application.properties` and YAML variants).
2. Profile-specific application properties packaged inside your jar (`application-{profile}.properties` and YAML variants).
3. Application properties outside of your packaged jar (`application.properties` and YAML variants).
4. Profile-specific application properties outside of your packaged jar (`application-{profile}.properties` and YAML variants).

NOTE	It is recommended to stick with one format for your entire application. If you have configuration files with both `.properties` and YAML format in the same location, `.properties` takes precedence.
NOTE	If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores instead (for example, `SPRING_CONFIG_NAME` instead of `spring.config.name`). See Binding From Environment Variables for details.
NOTE	If your application runs in a servlet container or application server, then JNDI properties (in `java:comp/env`) or servlet context initialization parameters can be used instead of, or as well as, environment variables or system properties.

To provide a concrete example, suppose you develop a `@Component` that uses a `name` property, as shown in the following example:

Java

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @Value("${name}")
    private String name;

    // ...

}
```

Kotlin

```
import org.springframework.beans.factory.annotation.Value
import org.springframework.stereotype.Component

@Component
class MyBean {

    @Value("\${name}")
    private val name: String? = null

    // ...

}
```

On your application classpath (for example, inside your jar) you can have an `application.properties` file that provides a sensible default property value for `name`. When running in a new environment, an `application.properties` file can be provided outside of your jar that overrides the `name`. For one-off testing, you can launch with a specific command line switch (for example, `java -jar app.jar --name="Spring"`).

TIP	The `env` and `configprops` endpoints can be useful in determining why a property has a particular value. You can use these two endpoints to diagnose unexpected property values. See the "Production ready features" section for details.

7.2.1. 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)`.

7.2.2. JSON Application Properties

Environment variables and system properties often have restrictions that mean some property names cannot be used. To help with this, 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:

```
$ SPRING_APPLICATION_JSON='{"my":{"name":"test"}}' java -jar myapp.jar
```

In the preceding example, you end up with `my.name=test` in the Spring `Environment`.

The same JSON can also be provided as a system property:

```
$ java -Dspring.application.json='{"my":{"name":"test"}}' -jar myapp.jar
```

Or you could supply the JSON by using a command line argument:

```
$ java -jar myapp.jar --spring.application.json='{"my":{"name":"test"}}'
```

If you are deploying to a classic Application Server, you could also use a JNDI variable named `java:comp/env/spring.application.json`.

NOTE	Although `null` values from the JSON will be added to the resulting property source, the `PropertySourcesPropertyResolver` treats `null` properties as missing values. This means that the JSON cannot override properties from lower order property sources with a `null` value.

7.2.3. External Application Properties

Spring Boot will automatically find and load `application.properties` and `application.yaml` files from the following locations when your application starts:

1. From the classpath
   a. The classpath root
   b. The classpath `/config` package
2. From the current directory
   a. The current directory
   b. The `config/` subdirectory in the current directory
   c. Immediate child directories of the `config/` subdirectory

The list is ordered by precedence (with values from lower items overriding earlier ones). Documents from the loaded files are added as `PropertySources` to the Spring `Environment`.

If you do not like `application` as the configuration file name, you can switch to another file name by specifying a `spring.config.name` environment property. For example, to look for `myproject.properties` and `myproject.yaml` files you can run your application as follows:

```
$ java -jar myproject.jar --spring.config.name=myproject
```

You can also refer to an explicit location by using the `spring.config.location` environment property. This property accepts a comma-separated list of one or more locations to check.

The following example shows how to specify two distinct files:

```
$ java -jar myproject.jar --spring.config.location=\
    optional:classpath:/default.properties,\
    optional:classpath:/override.properties
```

TIP	Use the prefix `optional:` if the locations are optional and you do not mind if they do not exist.

WARNING	`spring.config.name`, `spring.config.location`, and `spring.config.additional-location` are used very early to determine which files have to be loaded. They must be defined as an environment property (typically an OS environment variable, a system property, or a command-line argument).

If `spring.config.location` contains directories (as opposed to files), they should end in `/`. At runtime they will be appended with the names generated from `spring.config.name` before being loaded. Files specified in `spring.config.location` are imported directly.

NOTE	Both directory and file location values are also expanded to check for profile-specific files. For example, if you have a `spring.config.location` of `classpath:myconfig.properties`, you will also find appropriate `classpath:myconfig-<profile>.properties` files are loaded.

In most situations, each `spring.config.location` item you add will reference a single file or directory. Locations are processed in the order that they are defined and later ones can override the values of earlier ones.

If you have a complex location setup, and you use profile-specific configuration files, you may need to provide further hints so that Spring Boot knows how they should be grouped. A location group is a collection of locations that are all considered at the same level. For example, you might want to group all classpath locations, then all external locations. Items within a location group should be separated with `;`. See the example in the "Profile Specific Files" section for more details.

Locations configured by using `spring.config.location` replace the default locations. For example, if `spring.config.location` is configured with the value `optional:classpath:/custom-config/,optional:file:./custom-config/`, the complete set of locations considered is:

1. `optional:classpath:custom-config/`
2. `optional:file:./custom-config/`

If you prefer to add additional locations, rather than replacing them, you can use `spring.config.additional-location`. Properties loaded from additional locations can override those in the default locations. For example, if `spring.config.additional-location` is configured with the value `optional:classpath:/custom-config/,optional:file:./custom-config/`, the complete set of locations considered is:

1. `optional:classpath:/;optional:classpath:/config/`
2. `optional:file:./;optional:file:./config/;optional:file:./config/*/`

3. `optional:classpath:custom-config/`
4. `optional:file:./custom-config/`

This search ordering lets you specify default values in one configuration file and then selectively override those values in another. You can provide default values for your application in `application.properties` (or whatever other basename you choose with `spring.config.name`) in one of the default locations. These default values can then be overridden at runtime with a different file located in one of the custom locations.

Optional Locations

By default, when a specified config data location does not exist, Spring Boot will throw a `ConfigDataLocationNotFoundException` and your application will not start.

If you want to specify a location, but you do not mind if it does not always exist, you can use the `optional:` prefix. You can use this prefix with the `spring.config.location` and `spring.config.additional-location` properties, as well as with `spring.config.import` declarations.

For example, a `spring.config.import` value of `optional:file:./myconfig.properties` allows your application to start, even if the `myconfig.properties` file is missing.

If you want to ignore all `ConfigDataLocationNotFoundExceptions` and always continue to start your application, you can use the `spring.config.on-not-found` property. Set the value to `ignore` using `SpringApplication.setDefaultProperties(…)` or with a system/environment variable.

Wildcard Locations

If a config file location includes the `*` character for the last path segment, it is considered a wildcard location. Wildcards are expanded when the config is loaded so that immediate subdirectories are also checked. Wildcard locations are particularly useful in an environment such as Kubernetes when there are multiple sources of config properties.

For example, if you have some Redis configuration and some MySQL configuration, you might want to keep those two pieces of configuration separate, while requiring that both those are present in an `application.properties` file. This might result in two separate `application.properties` files mounted at different locations such as `/config/redis/application.properties` and `/config/mysql/application.properties`. In such a case, having a wildcard location of `config/*/`, will result in both files being processed.

By default, Spring Boot includes `config/*/` in the default search locations. It means that all subdirectories of the `/config` directory outside of your jar will be searched.

You can use wildcard locations yourself with the `spring.config.location` and `spring.config.additional-location` properties.

NOTE	A wildcard location must contain only one `*` and end with `*/` for search locations that are directories or `*/<filename>` for search locations that are files. Locations with wildcards are sorted alphabetically based on the absolute path of the file names.

TIP

Wildcard locations only work with external directories. You cannot use a wildcard in a `classpath:` location.

Profile Specific Files

As well as `application` property files, Spring Boot will also attempt to load profile-specific files using the naming convention `application-{profile}`. For example, if your application activates a profile named `prod` and uses YAML files, then both `application.yaml` and `application-prod.yaml` will be considered.

Profile-specific properties are loaded from the same locations as standard `application.properties`, with profile-specific files always overriding the non-specific ones. If several profiles are specified, a last-wins strategy applies. For example, if profiles `prod,live` are specified by the `spring.profiles.active` property, values in `application-prod.properties` can be overridden by those in `application-live.properties`.

NOTE

The last-wins strategy applies at the location group level. A `spring.config.location` of `classpath:/cfg/,classpath:/ext/` will not have the same override rules as `classpath:/cfg/;classpath:/ext/`.

For example, continuing our `prod,live` example above, we might have the following files:

```
/cfg
  application-live.properties
/ext
  application-live.properties
  application-prod.properties
```

When we have a `spring.config.location` of `classpath:/cfg/,classpath:/ext/` we process all `/cfg` files before all `/ext` files:

1. `/cfg/application-live.properties`
2. `/ext/application-prod.properties`
3. `/ext/application-live.properties`

When we have `classpath:/cfg/;classpath:/ext/` instead (with a `;` delimiter) we process `/cfg` and `/ext` at the same level:

1. `/ext/application-prod.properties`
2. `/cfg/application-live.properties`
3. `/ext/application-live.properties`

The `Environment` has a set of default profiles (by default, `[default]`) that are used if no active profiles are set. In other words, if no profiles are explicitly activated, then properties from `application-default` are considered.

NOTE	Properties files are only ever loaded once. If you have already directly imported a profile specific property files then it will not be imported a second time.

Importing Additional Data

Application properties may import further config data from other locations using the `spring.config.import` property. Imports are processed as they are discovered, and are treated as additional documents inserted immediately below the one that declares the import.

For example, you might have the following in your classpath `application.properties` file:

Properties

```
spring.application.name=myapp
spring.config.import=optional:file:./dev.properties
```

Yaml

```
spring:
  application:
    name: "myapp"
  config:
    import: "optional:file:./dev.properties"
```

This will trigger the import of a `dev.properties` file in current directory (if such a file exists). Values from the imported `dev.properties` will take precedence over the file that triggered the import. In the above example, the `dev.properties` could redefine `spring.application.name` to a different value.

An import will only be imported once no matter how many times it is declared. The order an import is defined inside a single document within the properties/yaml file does not matter. For instance, the two examples below produce the same result:

Properties

```
spring.config.import=my.properties
my.property=value
```

Yaml

```
spring:
  config:
    import: "my.properties"
my:
  property: "value"
```