spring-framework

Contents

- Introduction
  - Document Prerequisites
  - Spring and Spring boot
- Modern guidelines
  - 1. Use `record` for DTOs
  - 2. Prefer `var` for Local Variables (Judiciously)
  - 3. Replace Lombok with Java Language Features
  - 4. Sealed Classes for Domain Models
  - 5. Pattern Matching (`instanceof` and `switch`)
  - 6. Text Blocks for JSON/HTML/SQL
  - 7. Null Checks with `Objects.requireNonNullElse`
  - 8. HTTP Interface Clients (Java 21+)
  - 9. Avoid `@Autowired` use Constructor Injection Only
  - 10. Switch Expressions
- Getting started
  - Command line interface
  - Getting started
  - Packaging application
  - Build system
  - Best practices
    - Default packages
    - Typical layout
    - Auto-configuration
  - Application properties
    - Environment
    - Properties file
    - Value annotation
  - Spring Profiles
  - Spring Logging
    - SLF4J (Simple Logging Facade for Java)
    - Logback
    - Log4j2
    - java.util.logging (JUL)
  - Usage example
- Restful services
  - Request Mapping
  - Request Body
  - Path Variable
  - Request Parameter
  - Request Header

    - Request Attribute
    - Controller example
- Handling exceptions
    - @ControllerAdvice
    - @ExceptionHandler

Introduction

Spring boot is an open source java based framework used to create micro services, it is developed by Pivotal it is easy to create a stand alone and production read spring application, using spring boo. Spring Boot enterprise ready application that you can just run.

Document Prerequisites

This introduction is written to developers and readers who already have experience with Java, Spring, Maven and Gradle, you can easily understand the concepts of Spring Boot, if you have the aforementioned knowledge already. If you are a beginner we suggest you to go through some resources which serve as introduction to the topics and technologies mentioned above before you start reading this resource.

Spring and Spring boot

There are two different concepts here, which are Spring and Spring Boot. What is the difference between these ? Spring is the core framework, it consists of many components which are completely de-coupled from each other, they are standalone components used in building systems, they can be composed and layered on top of each other to build complex systems. However this requires a good amount of configuration to do, it is a complex process and it is more often than not very common to build applications of the same type very often. Therefore the same amount of work has to be done every time. This is where the Spring Boot comes in, the spring boot framework is also a list of components which are simply Spring components that are pre-configured to work together. These components are usually represented by Gradle or Maven dependencies, and to distinguish them from the regular spring components they are suffixed with the '-boot' work in their name, they are nothing more than a collection of raw Spring components that are made to work together, but in its core Spring Boot is simply just a well defined and configured common set of components for developing applications on top of the core Spring framework where all the cruft and complexity is abstracted into

Spring Boot automatically configures your application based on the dependencies you have added to the project by using `@EnableAutoConfiguration` annotation. For example, if MySQL database is on your `classpath`, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database. The entry point of the spring boot application is the class contains `@SpringBootApplication` annotation and the main method. Spring Boot automatically scans all the components included in the project by using `@ComponentScan` annotation.

As mentioned already all spring boot components contain the spring-boot name prefix, along with the starter word, which is simply a way of marking a component as part of the spring-boot framework, and the starter signals that this component is meant to be used as a bedrock for starting with the development, it should contain all the basics that one should expect for the dependency to work with spring and all of its internal components, without requiring a massive amount of user defined configuration to produce something useable

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

Here is a general idea of what an entry point looks like in a spring boot application, where we have the following annotation, `@SpringBootApplication`, this annotation alone is a combination of many other annotations, which are in charge of constructing the initial context for running your application.

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

@SpringBootApplication
public class DemoApplication {

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

Modern guidelines

1. Use `record` for DTOs

- Auto-generates `equals()`, `hashCode()`, `toString()`, and getters.
- Immutable by design (no Lombok needed).
- Request/response DTOs (`@RequestBody`, `@ResponseBody`).
- Immutable configuration properties (`@ConfigurationProperties`).

```
public class UserDto {
    private final String name;
    private final int age;
    // Boilerplate: constructor, getters, equals, hashCode, toString
}
```

```
public record UserDto(String name, int age) { }
```

2. Prefer `var` for Local Variables (Judiciously)

- Use `var` when the type is obvious (e.g., `new` expressions, builders).
- Avoid `var` if it reduces readability (e.g., `var result = service.process()`).

```
List<String> names = new ArrayList<>();
```

```
var names = new ArrayList<String>();
```

3. Replace Lombok with Java Language Features

- Use `record` for DTOs (replaces `@Data`, `@Value`).
- Use compact constructors:
- When to Keep Lombok:
  - `@Slf4j` (still concise).
  - `@Builder` (until Java gets a native builder pattern).

```
@Data
@Builder
public class Product { ... }
```

```
public record Product(String id, String name) {
    public Product {
        Objects.requireNonNull(id);
    }
}
```

4. Sealed Classes for Domain Models

- Explicitly restricts inheritance (better domain modeling).
- Works great with Spring Data JPA `@Entity` hierarchies.

```
public abstract class Shape { ... }
public class Circle extends Shape { ... }  // Unlimited extensibility
```

```
public sealed class Shape permits Circle, Rectangle { ... }
```

5. Pattern Matching (`instanceof` and `switch`)

- Cleaner controller logic (e.g., handling polymorphic DTOs).

```
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.length());
}
```

```
if (obj instanceof String s) {
    System.out.println(s.length());
}
```

6. Text Blocks for JSON/HTML/SQL

- `@Sql` annotations in internal tests.
- Hardcoded API response examples.

```
String json = "{\"name\":\"John\", \"age\":30}";
```

```
String json = """
    {
        "name": "John",
        "age": 30
    }
    """;
```

7. Null Checks with `Objects.requireNonNullElse`

```
return name != null ? name : "default";
```

```
return Objects.requireNonNullElse(name, "default");
```

8. HTTP Interface Clients (Java 21+)

- No need for `@FeignClient` (standard Java interface).

```
@FeignClient(url = "https://api.example.com")
public interface UserClient {
    @GetMapping("/users/{id}")
    User getUser(@PathVariable String id);
}
```

```
public interface UserClient {
    @GetExchange("/users/{id}")
    User getUser(String id);
}
```

9. Avoid `@Autowired` use Constructor Injection Only

- Immutable dependencies (thread-safe, no Lombok `@RequiredArgsConstructor` needed).

```
@Autowired
private UserService userService;
```

```
private final UserService userService;

public UserController(UserService userService) {
    this.userService = userService;
}
```

10. Switch Expressions

```
switch (status) {
    case "OK": return 200;
    case "BAD": return 400;
    default: return 500;
}
```

```
return switch (status) {
    case "OK" -> 200;
    case "BAD" -> 400;
    default -> 500;
};
```

Getting started

This section will teach you how to create a Spring Boot application using Maven and Gradle.

Command line interface

The Spring Boot CLI is a command line tool and it allows us to run the Groovy scripts. This is the easiest way to create a Spring Boot application by using the Spring Boot Command Line Interface. You can create, run and test the application in command prompt itself.

The spring cli has a neat way to generate projects, by simply providing the name of the project and some build and project arguments one can build any type of project skeleton very quickly, with the required set of dependencies and build environment conditions. This is mostly applicable for Maven and Gradle projects.

```
# this would build a very simple maven project skeleton, that would only
   include the basic spring boot starter and
# starter test dependencies and simple pom file for the maven project, as
   specified in the command below
$ spring init spring-boot --java-version=17 --build=maven
```

```
# here is the list, heavily abridged for demonstration purposes, that lists
   what are the supported dependencies, project
# types and, different project parameters one can pass to the spring cli to
   construct a project skeleton
$ spring init --list

Supported dependencies
+-----------------------------------------+-----------------------------------

| Id                                      | Description
                                                | Required version      |
+-----------------------------------------+-----------------------------------

| activemq                                | Spring JMS support with Apache
    ActiveMQ 'Classic'.           |                          |
|                                         |
                                                           |
                          |
| actuator                                | Supports built in (or custom)
   endpoints that let you monitor |                          |
|                                         | and manage your application -
   such as application health,    |                          |
|                                         | metrics, sessions, etc.
                                      |                          |
|                                         |
                                                           |
                          |
| amqp                                    | Gives your applications a
   common platform to send and        |                          |
|                                         | receive messages, and your
   messages a safe place to live     |                          |
|                                         | until received.
                                             |                          |
|                                         |
                                                           |
                          |
| azure-support                           | Auto-configuration for Azure
   Services (Service Bus, Storage, | >=3.4.0 and <3.5.0-M1 |
|                                         | Active Directory, Key Vault,
   and more).                     |                          |
|                                         |
                                                           |
```

batch		Batch applications with transactions, retry/skip and chunk based processing.
cache		Provides cache-related operations, such as the ability to update the content of the cache, but does not provide the actual data store.
wavefront	>=3.4.0 and <3.6.0-M1	Publish metrics and optionally distributed traces to Tanzu Observability by Wavefront, a SaaS-based metrics monitoring and analytics platform that lets you visualize, query, and alert over data from across your entire stack.
web		Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
web-services		Facilitates contract-first SOAP development. Allows **for** the creation of flexible web services using one of the many ways to manipulate XML payloads.
webflux		Build reactive web applications with Spring WebFlux and Netty.

```
                          |
|                                         |
                                                    |
                          |
| websocket                               | Build Servlet-based WebSocket
   applications with SockJS and   |                        |
|                                         | STOMP.
                                                  |
                          |
|                                         |
                                                    |
                          |
| zipkin                                  | Enable and expose span and
   trace IDs to Zipkin.              |                        |
+-----------------------------------------+-----------------------------------


Project types (* denotes the default)
+----------------------+-----------------------------------------------------

| Id                   | Description
                                          | Tags
                                  |
+----------------------+-----------------------------------------------------

| gradle-build         | Generate a Gradle build file.
                                 | build:gradle,format:build
                  |
|                      |
                                                    |
                                          |
| gradle-project *     | Generate a Gradle based project archive using the
   Groovy     | build:gradle,dialect:groovy,format:project |
|                      | DSL.
                                                   |
                                          |
|                      |
                                                    |
                                          |
| gradle-project-kotlin | Generate a Gradle based project archive using the
   Kotlin     | build:gradle,dialect:kotlin,format:project |
|                      | DSL.
                                                   |
                                          |
|                      |
                                                    |
                                          |
| maven-build          | Generate a Maven pom.xml.
                                    | build:maven,format:build
                   |
```

```
|                        |
                                                          |
                                          |
| maven-project          | Generate a Maven based project archive.
                         | build:maven,format:project                |
+------------------------+-------------------------------------------------------


Parameters
+-------------+-----------------------------------------+-----------------------
| Id          | Description                             | Default value
                |
+-------------+-----------------------------------------+-----------------------
| artifactId  | project coordinates (infer archive name) | demo
                           |
| bootVersion | spring boot version                      | 3.5.3
                          |
| description | project description                      | Demo project for
  Spring Boot |
| groupId     | project coordinates                      | com.example
                      |
| javaVersion | language level                           | 17
                             |
| language    | programming language                     | java
                            |
| name        | project name (infer application name)    | demo
                            |
| packageName | root package                             | com.example.demo
               |
| packaging   | project packaging                        | jar
                             |
| type        | project type                             | gradle-project
                 |
| version     | project version                          | 0.0.1-SNAPSHOT
                 |
+-------------+-----------------------------------------+-----------------------
```

Getting started

Spring boot provides a number of starters to add the jars in our class paths, for example for writing a rest endpoint we end to add the spring-boot-starter-web dependency in our class path. Observe the codes shown below for a better understanding.

The main method should be writing the Spring Boot Application class. This class should be annotated with the `SpringBootApplication` annotation, this is the entry point for the spring boot application to start, you can find the main class file under the src/java/main path / directory

Now to write a simple Hello World Rest endpoint in the spring boot application main class file itself, follow

the steps shown below:

- Firstly, add the `RestControoller` annotation at the top of the class
- Now, write a request URI method with the `RequestMethod` annotation
- Then the request URI method should return the Hello World string

```
package com.tutorialspoint.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class HelloWorldController {

    @RequestMapping(value="/")
    public String hello() {
        return "Hello World";
    }
}
```

Packaging application

Creating an executable JAR. Let us create and executable JAR file to run the Spring Boot Application, by using Maven or Gradle. The Spring executable JAR is somewhat different than regular JAR files which one can execute, the Spring `JARs` are meant to be self containing, they bundle the actual web server that is used to run the application, which could be a small web server like Tomcat, then on top of that all other dependencies are included into the jar. Unlike regular JARs (which only contain your classes in `/`), a Spring Boot executable JAR has:

First make sure that the maven plugin that is responsible for packaging the spring application is included into your pom file, we are not going to be using the regular maven package plugin - `maven-jar-plugin`

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.0</version>
</parent>
```

```
BOOT-INF/
     classes/      # Your compiled application classes
     lib/          # All dependency JARs
META-INF/
     MANIFEST.MF   # Defines the launcher class
org/springframework/boot/loader/
     JarLauncher.class  # Spring Boot's custom launcher
```

- Spring Boot uses `JarLauncher` (from `spring-boot-loader`) to **bootstrap** the application.
- This launcher knows how to load classes from `BOOT-INF/classes` and `BOOT-INF/lib`, unlike the standard Java classloader.
- All dependencies are **embedded** in the JAR (no need for an external `lib` folder).
- Avoids `classpath` issues when deploying.

How does it work ? If we compare the standard jar packaging plugin provided by maven to the spring one. The Standard maven plugin would package the JAR without including the dependencies or libraries that our project depends on, just the compiled classes for our project, the structure of a standard jar might look something like that

```
your-app.jar
 META-INF/
    MANIFEST.MF       # Basic metadata (Main-Class points directly to your
       app)
 com/
    yourpackage/      # Your compiled classes
 (No dependencies!)    # Requires external `lib/*.jar` files
```

Meaning we have no embedded dependencies included, and we have no embedded server included in the final JAR, that means that these have to be provided manually, that works if we are trying to deploy on an already existing web server, which presumable would have these dependencies installed. There are many ways to run a jar without its dependencies, but the most common one is to simply add those to the classpath, meaning that we have to build the command `java -jar` to include the required dependencies along with the jar we are trying to execute, this could be useful on systems where many jars are deployed on a single web server, and the dependencies are shared across the jars for the most part. This is how it is done in monolithic systems where micro services are not deployed.

```
# first we copy all the dependencies into a common directory, which would
   then be passed to the java cp command, note
# that we are using shell expansion here, the target/lib/*, is shell specific
    and the shell itself would expand all the
# jars in the lib folder when running the command, this is not something that
    is done or understood by the java binary,
mkdir -p target/lib && cp myapp.jar target/lig
mvn dependency:copy-dependencies -DoutputDirectory=target/lib
java -cp "target/your-app.jar:target/lib/*" com.myapp.MainClass
```

Note that we are passing the entire classpath to the JVM, above, we are not using `java -jar` that is because that command expects the jar to have a valid MANIFEST file, which the jar compiled and packaged by default from the maven-jar-plugin does not. We are going to leverage another plugin called the assembly plugin to do exactly that - build a MANIFEST file into the jar which will be read by the JVM to construct the classpath based on the contents of that file

There are also other ways to do this by configuring the maven plugin to package these libraries for you while the jar is being built, this is what the spring boot plugin does actually it would put a lib/ folder with the libraries and dependencies directly into the jar. Which is effectively the same as passing it on the classpath. Here is an example of how you can package the jar using the assembly plugin which would effectively do what we discussed

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.6.0</version>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
```

```
            <!-- Here is the config for the MANIFEST file that will be
                constructed and put into the jar -->
            <manifest>
                <mainClass>com.yourpackage.MainClass</mainClass>
            </manifest>
        </archive>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

```
# after the assembly plugin has done its job, we would have a jar file which
   can be simply run like so, which is pretty
# much the same way we run the spring jars as well, since your jar would now
   contain the MANIFEST file, which described
# how to construct the classpath and where to find all the libraries and
   dependencies required
$ java -jar app.jar
```

So how does the spring boot maven plugin package the jar and how does it run, the following steps are being followed to execute this process:

1. **We start by running the jar**:

   ```
   $ java -jar your-app.jar
   ```

2. **Java virtual machine reads `META-INF/MANIFEST.MF`**:

   ```
   Manifest-Version: 1.0
   Created-By: Maven JAR Plugin 3.4.2
   Build-Jdk-Spec: 17
   Implementation-Title: demo
   Implementation-Version: 0.0.1-SNAPSHOT
   Main-Class: org.springframework.boot.loader.launch.JarLauncher
   Start-Class: com.spring.demo.DemoApplication
   Spring-Boot-Version: 3.5.3
   Spring-Boot-Classes: BOOT-INF/classes/
   Spring-Boot-Lib: BOOT-INF/lib/
   Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx
   Spring-Boot-Layers-Index: BOOT-INF/layers.idx
   ```

3. **`JarLauncher` takes over**:

   - Sets up a **custom classloader** to load classes from:
     - `BOOT-INF/classes/` (your code).
     - `BOOT-INF/lib/*.jar` (dependencies).
   - Discovers and launches the `Start-Class` (your `@SpringBootApplication` class).

4. **Spring Boot initializes**:
    - Starts the embedded server (if applicable).
    - Autoconfigures beans, initialize the context, etc.

So if we want to really compare both here is what we have established so far, the main differences between the maven-jar-plugin and the spring-boot-maven-plugin.

Feature	`maven-jar-plugin`	`spring-boot-maven-plugin`
Dependencies	Not included (external `lib/`)	Embedded in `BOOT-INF/lib/`
Main-Class	Directly points to your class	Delegates to `JarLauncher`
Class Loading	Standard JVM classloader	Custom `LaunchedURLClassLoader`
Runnable	Requires manual classpath setup	Works with `java -jar`
Web Server	Needs external Tomcat/Jetty	Embedded server (Tomcat/Netty/etc.)

There is another type of JAR archive, called a WAR application, this is another type of JAR that is deployed on a web server, unlike the JAR file, the WAR has a different format and specification this is because the web server where this WAR will be deployed requires a specific format and structure to be present.

A **WAR** is a specialized JAR used for **web applications** (e.g., Servlet-based apps like Spring MVC). It follows a strict structure defined by the `Java EE/Jakarta EE` spec and is deployed to **external servers** (e.g., Tomcat, Jetty, WildFly).

```
my-app.war
 WEB-INF/
    classes/       # Your compiled classes (e.g., `com.yourpackage`)
    lib/           # Dependency JARs (external to the WAR)
    web.xml        # Servlet configuration (optional in modern apps)
 META-INF/          # Metadata (e.g., context info)
 static/            # Static files (HTML, JS, CSS)
```

Requires a Servlet container (e.g., Tomcat). The dependencies in `WEB-INF/lib/` are loaded by the server, and there is no main class, the server starts the app (not `java -jar`).

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.4.0</version>
</plugin>
```

To setup and deploy the application you can simply do

```
$ mvn clean package  # Generates `target/my-app.war`
$ cp target/my-app.war /opt/tomcat/webapps/
# Tomcat unpacks the WAR and starts the app.
```

To be able to share common libraries, we can use the special directory that tomcat checks, where we can Place common JARs (e.g., Spring, Hibernate, JDBC drivers) in the server's `lib/` folder. All deployed WARs automatically inherit these JARs in their classpath.

```
/opt/tomcat/lib/
 spring-core.jar
 hibernate-core.jar
 postgresql-driver.jar
```

We can also make sure to exclude the dependency from the final archive, by setting the provided property on the scope tag, this will signal the maven builder that this dependency must not be packaged in, and it will be `provided` by an external party, in this case it will be the web server - tomcat in our example.

```
<project>
  <packaging>war</packaging>
  <dependencies>
    <!-- Mark shared dependencies as 'provided' -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>6.1.0</version>
      <scope>provided</scope>  <!-- Server will supply this -->
    </dependency>
  </dependencies>
</project>
```

WARs can **omit libraries** if the server provides them centrally (via `lib/` or `shared/lib/`). This is common in traditional Java EE deployments but sacrifices version flexibility. Modern apps (especially Spring Boot) prefer **self-contained JARs** to avoid conflicts, and it is much harder to setup a micro service environment where the web server is separately deployed from the application, and the web server has the common dependencies already installed. This is more suitable for monolithic applications where there are centralized servers installed on big fat environments, and they can be pre-loaded with all the common dependencies marked as provided in the pom file of the application.

To understand it better, here are the different types of scopes that maven allows you to specify for a dependency, you can certainly see that there are different use cases, and in most use cases, we are either going to be using runtime, compiled or provided. For an embedded server, the most common one will be the compile one, since all the libraries must be present at compile time into the JAR, for WAR types we might make use of a lot of provided dependencies such as Tomcat shared implementation libraries that are going to be provided by the web server by default

Scope	Compile	Test	Runtime	Packaged	Example Use Case
`compile`					Spring, Hibernate
`provided`					Servlet API, Tomcat-shared libs
`runtime`					JDBC drivers
`test`					JUnit, Mockito
`system`					Avoid (non-portable JARs)
`import`					BOMs for dependency management

1. **`compile` (Default)**
   - Available in all phases (compile, test, runtime).
   - Packaged in the final JAR/WAR.
   - **Example**: `spring-core` (needed at runtime).
2. **`runtime`**
   - Not needed for compilation, but required at runtime.
   - Packaged in the final artifact.
   - **Example**: `mysql-connector-java` (loaded dynamically).
3. **`provided`**
   - Available during compilation and testing, but **not packaged**.
   - Assumed to be provided by the runtime (e.g., a server).

- **Example**: `javax.servlet-api` (provided by Tomcat).

4. `system` (Avoid if possible)
   - Like `provided`, but points to a local JAR file.
   - **Example**: Rarely used (non-portable).
5. `import` (Special case)
   - Used in `<dependencyManagement>` to inherit versions from another POM.
   - **Example**: `spring-boot-dependencies` (BOM files).
6. `test`
   - Only available during testing (`src/test`).
   - Never packaged.
   - **Example**: `junit` (test-only dependency).

Build system

In Spring Boot, choosing a build system is an important task. We recommend Maven or Gradle as they provide a good support for dependency management. Spring does not support well other build systems.

Spring Boot team provides a list of dependencies to support the Spring Boot version for its every release. You do not need to provide a version for dependencies in the build configuration file. Spring Boot automatically configures the dependencies version based on the release. Remember that when you upgrade the Spring Boot version, dependencies also will upgrade automatically. This is achieved by the parent BOM spring dependency which you specify, the BOM - which stands for bill of materials, is a set of pre-defined list of dependencies. The spring BOM has a version, which carries with it the correct combination of all spring dependency versions.

Best practices

Spring Boot does not have any code layout to work with. However, there are some best practices that will help us. This chapter talks about them in detail.

Default packages

A class that does not have any package declaration is considered as a **default package**. Note that generally a default package declaration is not recommended. Spring Boot will cause issues such as malfunctioning of Auto Configuration or Component Scan, when you use default package.

Typical layout

It is generally accepted that at the root name of the artifact id for the project - `com.myapp.project`, we should have the Application.java file, which contains the Main class, this is due to the fact that any class annotated with the `@SpringBootApplication` annotation, will serve as entry point for scanning the packages and beans under it, in that case if we put the main class at `com.myapp.project`, then everything under this root package name will be considered for injection by the spring components.

Auto-configuration

As mentioned in spring we have a concept called an Auto configuration, this is a process by which spring picks specific pre-defined beans on the class path. The way it works is by defining a special type of file that contains the full path to the class that is considered to be the auto-configuration class, this file is called the `org.springframework.boot.autoconfigure.AutoConfiguration.imports`, this file contains lines of text which define the different classes spring autoconfigure should consider when starting the application, this is different compared to classes annotated with @Configuration, which we will talk about later.

Here is the abridged version of the file included in the `META-INF/spring/` folder in the autoconfigure dependency, this file is automatically picked when we define the `EnableAutoConfiguration` annotation on a spring project, then each of those classes will be instantiated and their methods run.

```
org.springframework.boot.autoconfigure.admin.
   SpringApplicationAdminJmxAutoConfiguration
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration
.....................................................................

org.springframework.boot.autoconfigure.websocket.reactive.
   WebSocketReactiveAutoConfiguration
org.springframework.boot.autoconfigure.websocket.servlet.
   WebSocketMessagingAutoConfiguration
org.springframework.boot.autoconfigure.websocket.servlet.
   WebSocketServletAutoConfiguration
```

It is prudent to note that you can also create your own imports file, by simply putting in tin the src/-main/resources folder of your own library of project, then when it gets built it will be put into the META-INF folder in your final JAR, that file will then be looked up by the spring autoconfigure library. Create the file into the resources directory `src/main/resources/META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`: Loaded **automatically** when the JAR is on the classpath, and no `scanBasePackages` required.

```
com.yourlib.autoconfig.CustomSecurityAutoConfig
```

```
package com.yourlib.autoconfig;

@Configuration
@ConditionalOnClass(SomeDependency.class)
public class CustomSecurityAutoConfig { ... }
```

On another hand we have the @Configuration annotation, that is a bit different since it is discovered through component scanning, which is a different process in spring, it also occurs in a different stage of the application startup process, this is later than the auto configuration classes as well. They are loaded if these fall under the scan base packages list, i.e. the class is in a package scanned by Spring (`com.yourpackage`), or explicitly imported with the annotation - `@Import(AppConfig.class)`.

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() { return new MyService(); }
}
```

```
In the end the result of both is the same, both types of classes are defined using the
same annotations, and the same rules, the only difference is that the spring provided auto
 configurations are mostly located in the starter packages, which spring provides, and are
 defined in the imports file, in theory if you remove the imports file, and put the spring
 org.srpingboot base package in the scanBasePackages annotation value of ComponentScan,
you will achieve the same effect, the only difference is that the configurations defined
 in the imports file are run earlier than the component scanning that spring does on the
 classes under packages defined in scanBasePackages, but that still happens after the
spring context is initialized
```

The general rule of thumb, is that if you are creating a library or re-usable component it is easier and cleaner to provide your own imports file, this would avoid having your package's users or clients to have to include the base package of your library or component every time into the `ComponentScan.scanBasePackages annotation`.

Application properties

Application Properties support us to work in different environments. In this section, you are going to see how to configure and specify the properties to a Spring Boot application.

The general order of properties collection that Spring Boot does is by evaluating sources in this order (higher overrides lower order):

1. **Command-line arguments** (`--key=value`, directly run the application with these properties).
2. **`SPRING_APPLICATION_JSON`** (environment variable with JSON properties, not a file location).
3. **`JNDI`** (Java Naming and Directory Interface).
4. **Externalized files** (`--spring.config.location`).
5. **Environment YAML/properties** (e.g., `application-prod.yml`).
6. **Default files `application.yml/application.properties`**.

Here an example of implementing the different types of property overrides when working with a spring application, also defined in order of most precedence to the least precedence.

1. **Command-line arguments** (`--key=value` in `java -jar`).

   ```
   java -jar app.jar --server.port=8081
   ```

2. **Environment variables** (e.g., `SPRING_DATASOURCE_URL`, `SERVER_PORT`).

   ```
   export SERVER_PORT=8081
   ```

3. **System properties** (`-Dkey=value` passed to JVM).

   ```
   java -Dserver.port=8081 -jar app.jar
   ```

4. **Profile-specific YAML/properties** (e.g., `application-prod.yml`).

5. **Default `application.yml/application.properties`**. ### Command line

Spring Boot application converts the command line properties into Spring Boot Environment properties. Command line properties take precedence over the other property sources. By default, Spring Boot uses the 8080 port number to start the Tomcat. Let us learn how change the port number by using command line properties. There are a few ways to override certain properties on the command line, such that they will take precedence over the ones with which the app was packaged

```
# Values will override any `server.port` or `spring.datasource.url` defined
   in `application.yml`.
$ java -jar your-app.jar --server.port=8081 --spring.datasource.url=jdbc:
   mysql://localhost:3306/mydb

# Properties in `custom-config.yml` will **override** those in `application.
   yml`.
# Add `classpath:` for files in the classpath (e.g., `classpath:override.
   properties`).
$ java -jar your-app.jar --spring.config.location=file:/path/to/custom-config
   .yml
```

```
# Merge additional files without replacing the packaged `application.yml`:
$ java -jar your-app.jar --spring.config.additional-location=file:/path/to/
   extra.properties
```

Environment

There are several ways to override the properties through the environment,

Core Spring Boot Variables

Env Variable	Effect
`SPRING_APPLICATION_JSON`	Injects properties as JSON (e.g., `{"server":{"port":8081}}`).
`SPRING_CONFIG_LOCATION`	Overrides config file path (e.g., `file:/config/override.yml`).
`SPRING_CONFIG_ADDITIONAL_LOCATION`	Adds extra config files without replacing defaults.
`SPRING_PROFILES_ACTIVE`	Sets active profiles (e.g., `prod,debug`).
`SPRING_CLOUD_CONFIG_ENABLED`	Disables/configures Spring Cloud Config (e.g., `false`).

Logging and Debugging

Env Variable	Effect
`DEBUG`	Enables debug logs if set to `true` or `1`.
`LOGGING_LEVEL_ROOT`	Sets root log level (e.g., `DEBUG`, `INFO`).
`LOGGING_LEVEL_org.springframework.web`	Sets package-specific logging (e.g., `DEBUG`).
`SPRING_OUTPUT_ANSI_ENABLED`	Force ANSI color output (e.g., `ALWAYS`).

Server and Database Overrides

Env Variable	Effect
`SERVER_PORT`	Overrides `server.port` (e.g., `8081`).
`SERVER_SERVLET_CONTEXT_PATH`	Sets the context path (e.g., `/api`).
`SPRING_DATASOURCE_URL`	Overrides DB URL (e.g., `jdbc:mysql://db:3306/app`).
`SPRING_DATASOURCE_USERNAME`	Sets DB username.
`SPRING_DATASOURCE_PASSWORD`	Sets DB password.

Replace `.` with `_` and uppercase (e.g., `server.port` → `SERVER_PORT`). Nested keys: `spring.datasource.url` → `SPRING_DATASOURCE_URL`.

Overriding with `JAVA_TOOL_OPTIONS` If you can't use Spring-specific vars, pass the Java system properties, which will be picked up by spring, since it does a lookup in order of environment variables, system variables, and then proceeds with startup arguments

```
export JAVA_TOOL_OPTIONS="-Dserver.port=8081 -Ddebug=true"
```

Properties file

Properties files are used to keep 'N' number of properties in a single file to run the application in a different environment. In Spring Boot, properties are kept in the **application.[properties/yml]** files under the classpath. The application.properties file is located in the **src/main/resources** directory. The code for sample **application.properties** and the **application.yml** file are given below:

```
server.port=9090
spring.application.name=demoservice
```

```
spring:
 application:
    name: demo
server:
 port: 8080
```

Value annotation

The @Value annotation is used to read the environment or application property value in Java code. The syntax to read the property value is shown below:

```
@Value("${spring.application.name}")
private String applicationName;
```

Spring Profiles

Spring boot supports different properties based on the Spring active profile. For example we can keep two separate files for development and production to run the Spring Boot application, or we can also use multi-document setup in the same file, in a regular yml file it is possible to separate different documents using the --- atom, which is the equivalent of having different / separate files.

```
---
# make sure that the logging is active for all profiles by default, we are
   using the INFO level, which means that we
# will be logging everything above and including INFO level, that implies -
   INFO, WARN, ERROR.
logging:
  level:
    com.spring.demo.core: INFO

# this is the base document, or the default one which will be picked up by
   spring, anything defined here can be
# overridden with specific active profile configuration
spring:
  application:
    name: demo-base

---
spring:
  application:
    name: demo-local
  config:
    activate:
```

```
      # this tells spring that this document will be only active if the
         current profile that is active is called `local`, in
      # that case this file / document will be merged with the base config
      on-profile:
      - local
---
```

The other traditional option that spring provides is separate files, each file has to include the name of the active profile in its name - `application, application-local, application-dev`, where the active properties will be, respectively in order - the profiles `default , local and dev`. The default profile is always active.

While running the JAR file, we need to specify the spring active profile based on each properties file. By default, Spring Boot application uses the application.properties file. The command to set the spring active profile is shown below:

```
$ java -jar <demo-application>.jar --spring.profiles.active=dev
```

```
2017-11-26 08:13:16.322 INFO 14028 --- [ main]
com.tutorialspoint.demo.DemoApplication : The following profiles are active:
dev
```

Spring Logging

Spring Boot uses Apache Commons logging for all internal logging. Spring Boot's default configurations provides a support for the use of Java Util Logging, Log4j2, and Logback. Using these, we can configure the console logging as well as file logging.

If you are using Spring Boot Starters, Logback will provide a good support for logging. Besides, Logback also provides a use of good support for Common Logging, Util Logging, Log4J, and SLF4J.

The default log messages will print to the console window. By default, "INFO", "ERROR" and "WARN" log messages will print in the log file.

```
[INFO] +- org.springframework.boot:spring-boot-starter-logging:jar:3.5.3:
   compile
[INFO] |  +- ch.qos.logback:logback-classic:jar:1.5.18:compile
[INFO] |  |  +- ch.qos.logback:logback-core:jar:1.5.18:compile
[INFO] |  |  \- org.slf4j:slf4j-api:jar:2.0.17:compile
[INFO] |  +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.24.3:compile
[INFO] |  |  \- org.apache.logging.log4j:log4j-api:jar:2.24.3:compile
[INFO] |  \- org.slf4j:jul-to-slf4j:jar:2.0.17:compile
```

SLF4J (Simple Logging Facade for Java)

- **Role**: Abstraction layer (facade) over concrete logging implementations.
- **Purpose**: Decouples your code from specific logging frameworks.
  - Provides a unified API (e.g., `LoggerFactory.getLogger()`).
  - Allows switching implementations (Logback, Log4j2, etc.) without code changes.

Logback

- **Default in Spring Boot**.
  - Native SLF4J implementation (no adapter needed).
  - Fast, flexible configuration via `logback.xml`.

Log4j2

- **Successor to Log4j 1.x**.
- **Strengths**:
  - High performance (asynchronous logging).
  - Advanced features (JSON/YAML config, plugins).

java.util.logging (JUL)

- **Built into the JVM**.
  - Limited features, poor performance.
  - Rarely used directly (SLF4J bridges exist).

```
import org.slf4j.Logger;
Logger logger = LoggerFactory.getLogger(MyClass.class); // SLF4J API
logger.info("Hello"); // Delegates to the implementation Logback/Log4j2
```

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-logging</artifactId> <!-- Logback -->
</dependency>
```

To use the `log4j2` Replace the starter dependency with `spring-boot-starter-log4j2`.

By default, all logs will print on the console window and not in the files. If you want to print the logs in a file, you need to set the property **logging.file** or **logging.path** in the application.properties file.

You can specify the log file path using the property shown below. Note that the log file name is spring.log.

```
# we have to provide the name of the log file, and path, by default these
   values are not set, thus no file logging is
# performed but it is a good practice in production to make sure both are
   explicitly provided, and have a persistent log location,
# note that this is using a relative path, the file will be created in the
   current working directory where the jar was run, the
# reason being is that depending on your system, paths like /tmp or /var
   might not be accessible to the process running the jar,
# and the log file will never be created, so make sure that your process has
   the permissions to read/write at the file.path location
logging.file.path=./tmp/
logging.file.name=spring.log
```

To modify the log levels, for all packages we can use the `root` prefix, this will enable logging levels for every package in the classpath, however it is possible to provide a more granular logging based on a specific package or even an entire sub-package.

```
logging:
    level:
        root=ERROR
        com.my.package=TRACE
        org.springframework.web: DEBUG
        org.hibernate.SQL: DEBUG
        org.hibernate.type: TRACE
```

Usage example

Here is a more advanced and complete example using the different options we have looked at such as logging, configurations, properties and injection

```
package com.spring.demo.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DefaultDemoAutoConfiguration {

    private static final Logger LOGGER = LoggerFactory.getLogger(
       DefaultDemoAutoConfiguration.class);

    public DefaultDemoAutoConfiguration() {
        LOGGER.info("Creating the auto configuration bean
           DefaultDemoAutoConfiguration");
    }

    @Bean
    DefaultAutoConfigBean defaultBeanFromAutoConfig(@Value("${spring.
       application.name}") String name) {
        LOGGER.info("Initializing DefaultAutoConfigBean");
        return new DefaultAutoConfigBean(name);
    }

    public static final class DefaultAutoConfigBean {

        public DefaultAutoConfigBean(String name) {
            LOGGER.info("Running DefaultAutoConfigBean constructor for {}",
               name);
        }
    }
}
```

Restful services

Before you proceed to build a RESTful web service, it is suggested that you have knowledge of the following annotations: The `@RestController` annotation is used to define the RESTful web services. It serves JSON, XML and custom response. Its syntax is shown below:

```
@RestController
public class ProductServiceController {
}
```

Request Mapping

The `@RequestMapping` annotation is used to define the Request URI to access the REST Endpoints. We can define Request method to consume and produce object. The default request method is GET. However the annotation itself has a field that can be used to specify the type - GET, POST, DELETE etc. All major METHOD types are supported out of the box. There are dedicated annotations which are called `PostMapping`, `GetMapping`, `DeleteMapping` and so forth, which are preferred over the plain direct usage of the `RequestMapping` annotation

```
@RequestMapping(value="/products")
public ResponseEntity<Object> getProducts() { }
```

Request Body

The `@RequestBody` annotation is used to define the request body content type. This is relevant when the request method in question is of type that is different than GET, in those cases it is possible to send payload to the Servlet. The payload in this case will be automatically de-serialized into the Product POJO, however if there are types mismatch present between the types of the fields or properties of the Product POJO then an error would be thrown.

```
@RequestMapping(method=RequestMethod.POST, value="/products")
public ResponseEntity<Object> createProduct(@RequestBody Product product) {
}
```

This example shows how we can read in an ambiguous set of values and fields from the request instead of having a dedicated java class that defines the fields, this is possible due to the usage of the Object in the map which is polymorphic at runtime. The de-serialized values in the Map would Object instances, but actually be de-serialized values of class primitives such as Integer, String, Byte, Boolean etc.

```
@RequestMapping(method=RequestMethod.POST, value="/products2")
public ResponseEntity<Object> createProductMap(@RequestBody Map<String,
   Object> product) {
}
```

Path Variable

The `@PathVariable` annotation is used to define the custom or dynamic request URI. The Path variable in request URI is defined as curly braces {} as shown below. This would inject the value of the templated entry in the path string in this case called {id} into the method argument, when the request is received, which would allow you to dynamically read a request value from the path. Take a good note of the name of the argument in the annotation - `id`, this has to match the value in the brackets. It is also possible to skip the value in the `@PathVariable` annotation, and the name would be inferred from the name of the argument of the function, this however is only possible if the jar is compiled with the -parameter option passed to the compiler, that would tell the compiler to retain the name of the arguments at runtime, making them accessible to the annotation processor that injects the value into the arguments of the function, by reading their names

```
@RequestMapping(method=RequestMethod.GET, value="/products/{id}")
public ResponseEntity<Object> updateProduct(@PathVariable("id") String id) {
}
```

Request Parameter

The `@RequestParam` annotation is used to read the request parameters from the Request URL. By default, it is a required parameter. We can also set default value for request parameters as shown here:

```
// one would be able to call this GET endpoint such as follows schema://
   hostname/products?name=product-name, the query
// parameter is optional meaning that a call will match without it being
   present, and a value will be defined by default if
// there is no value provided
@RequestMapping(method=RequestMethod.GET, value="/products")
public ResponseEntity<Object> getProduct(@RequestParam(value="name", required
   =false, defaultValue="honey") String name) {
}
```

Request Header

While a request is being built by the client it is possible to construct a custom header that can be passed in the header section of the HTTP payload, this header has a name and a value, in our case the name is `x-product-name` note that the header names are **`not case sensitive`**, the values in there are usually primitive strings, numbers or boolean types.

```
@RequestMapping(method=RequestMethod.GET, value="/products")
public ResponseEntity<Object> getProduct(@RequestHeader("X-Product-Name")
   String name) {
}
```

Request Attribute

The request attribute, is an internal construct that is attached to the request by the Servlet processor, as early as the request being captured by the web server, usually in the real world a special filter or interceptor might add some request attribute to the request before it reaches the method call in the restful controller, this can then be used to do some further processing, this usually can happen based on some business logic, or in the example below, based on some other properties found in the http request sent by the client.

```
// based on some business logic a special attribute might be added to the
   request before it reaches the controller method
// in our example we `imagine` that a special discount value will be added
   based on some heuristic calculated from the
// client request payload.
@RequestMapping(method=RequestMethod.GET, value="/products")
public ResponseEntity<AuditLog> getProducts(@RequestAttribute("discount")
   Double discount) {
}
```

Controller example

```
@RestController
public class ProductsController {

    @PostMapping("/products-body")
    public ResponseEntity<Object> getProductsQuery(@RequestBody
       ProductResourceRecord product) {
        return ResponseEntity.of(Optional.ofNullable(product));
    }
```

```
    @GetMapping("/products-header")
    public ResponseEntity<Object> getProductsHeader(@RequestHeader("name")
       String name) {
        return ResponseEntity.of(Optional.ofNullable(name));
    }

    @GetMapping("/products-query")
    public ResponseEntity<Object> getProductsQuery(@RequestParam("name")
       String name) {
        return ResponseEntity.of(Optional.ofNullable(name));
    }

    @GetMapping("/products-path/{id}")
    public ResponseEntity<Object> getProductsPath(@PathVariable("id") String
       id) {
        return ResponseEntity.of(Optional.ofNullable(id));
    }

    @GetMapping("/products-discount")
    public ResponseEntity<Object> getProductsDiscount(@RequestAttribute("
       discount") Double discount) {
        return ResponseEntity.of(Optional.ofNullable(discount));
    }
}
```

```
### 1. POST with @RequestBody
POST http://localhost:8080/products-body
Content-Type: application/json

{
    "id": "laptop",
    "name": "Laptop",
}

### 2. GET with @RequestHeader
GET http://localhost:8080/products-header
name: PremiumCustomer

### 3. GET with @RequestParam
GET http://localhost:8080/products-query?name=Smartphone

### 4. GET with @PathVariable
GET http://localhost:8080/products-path/12345

### 5. GET with @RequestAttribute (requires server-side setup)
GET http://localhost:8080/products-discount
```

Handling exceptions

Handling exceptions and errors in API, and sending the proper responses to the client is good for enterprise application, in this chapter we will learn how to handle the exceptions in spring boot, before proceeding with exception handling let us gain an understanding on the following annotations:

@ControllerAdvice

This annotation is used to handle the exceptions globally. What it does it create an advice - an advice is a type of functionality that is wrapped around a certain functionality, when you a controller method is triggered, a certain code can be executed before or after or around the execution of this method.

```
public ResponseEntity<Object> getProducts() {}
```

If we assume we have this method present in our controller, the Advice can be called right before the method is called, right after the method exits or/and both could be done as well. Meaning that we can do some generic work before or/and after the method execution.

In the most general use case that can be used to catch exceptions in a single location no mater which controller method was called. This is useful when used in conjunction with the `@ExceptionHandler` annotation.

@ExceptionHandler

This annotation is used to handle the specific exceptions, what it does is annotate a method inside a class annotated with `@ControllerAdvice`, that method will be called when a specific exception of a specific type is emitted from a controller method.

The advice is usually implemented internally with a library like `aspectj`, which allows us to inject at either runtime (through a proxy object) or compile time (by literally writing byte code during compile time) additional code around a certain construct like - method, constructor etc. The `ControllerAdvice` and the `ExceptionHandler` are provided by Spring to simplify the process.

```
// we create an advice for controllers, this annotation will pick up this
   class and create an advice handler wrapped
// around all methods in classes marked with RestController in our module, it
    could look something like that, in pseudo code:
// try {
//     return productController.getProducts();
// } catch(Throwable e) {
//     for handler in handlers {
//         if e instanceOf handler.getException() {
//             handler.executeExceptionHandlerMethod();
//         }
//     }
// }
@ControllerAdvice
public class ProductExceptionController {

    @ExceptionHandler(value = ProductNotfoundException.class)
    public ResponseEntity<Object> exception(ProductNotfoundException
       exception) {
        // handle the specific exception, if another exception is triggered
           this method will not be called, the
```

```
        // ExceptionHandler specifically handles only this type of exception
           and ignores others, if you wish to be more
        // generic you could create another method, annotated with -
           @ExceptionHandler(value = RuntimeException.class)
    }
}
```