**Introduction to Spring & Spring Boot**

* Spring boot is built up on Spring Framework
* Spring Framework was first introduced in 2003 by Rod Johnson
* Before 2003 developers were creating Java Application using Java Enterprise Edition
* The problem with JEE was, they had to configure a lot before they could write a single piece of business logic
* Configurations like Server, DB and logs
* Spring Framework helped developers configure all that
* The part of Spring Framework doing all those configurations is the IoC container. It manages the lifecycle and dependencies of the objects in a Spring application
* IoC uses XML files to configure all that
* IoC containers were still taking a lot of time in configurations
* In 2009 Rod Johnson sold spring to VMware
* In 2014 through VMware, came the Spring boot, which was already configured with starter dependencies and ready for business logic.

**Spring Framework**

* Spring is a Dependency Injection framework to make Java applications loosely coupled
* Spring framework makes the development process easy for JavaEE applications
* Spring enables you to build applications from “plain old Java objects” (POJO) and to apply enterprise services non-invasively to POJOS.
* Important Components
  + Core Container
  + AOP
  + JDBC
  + Web
  + Testing

**IoC Container**

* In the Spring framework, IoC container is responsible for managing the components of an application and injecting dependencies into them. The container creates the objects (beans), wires them together, configure them, and manages their complete lifecycle.
* IoC takes care of the event driven programming (EDP), Dependency Injection (DI), Aspect Oriented Programming (AOP)
* **IoC Container**:
  + The **Inversion of Control (IoC) container** is **responsible for instantiating, configuring, and assembling beans**.
  + It starts working **as soon as the application context is loaded**, either via XML configuration, annotations, or Java-based config.
* **Dependency Injection (DI)**:
  + The process where the IoC container **injects dependencies into beans** based on the configuration (e.g., via XML).
  + DI **happens during the initialization phase** of the container, **not before**.
* **POJOs (Business Objects)**:
  + These are the actual beans managed by the IoC container.
  + They are **instantiated and injected by the container** as it processes the configuration.

**Gradle, Maven and Artifact**

* In the Spring framework, Gradle and Maven are build automation tools sued to manage dependencies, built the project, and automate tasks like testing, packaging and deployment.
* The **artifact** is the name of the **build output** (e.g., myapp.jar).
* It also sets the **name of your project directory**, **application class**, and sometimes the **package structure**.
* It's part of the **Maven coordinate system** (groupId:artifactId:version).

**What Are Gradle and Maven?**

| **Feature** | **Maven** | **Gradle** |
| --- | --- | --- |
| Language | Uses **XML** (pom.xml) | Uses **Groovy/Kotlin DSL** (build.gradle or build.gradle.kts) |
| Performance | Slower builds due to XML parsing | Faster with **incremental builds** and **daemon** process |
| Popularity | Very mature and widely used | Newer, but increasingly popular |
| Configuration | More rigid and convention-based | More flexible and scriptable |
| Dependency Management | Based on a centralized repository model | Also uses Maven Central, but with more customization |

* POM stands for Project Object Model file

**Beans**

* Beans build up our spring applications
* Beans are the classes. Their cycles and dependencies are managed by the spring framework.
* Beans are a managed object that is instantiated, assembled, and managed by the spring IoC container
* Beans are the backbone of any spring application and they are the core building blocks that are wired together to create a application.
* They are used to manage our dependency framework

**Spring Annotations**

* Traditionally, Spring allows a developer to manage bean dependencies by using XML-based configuration
* With Spring Boot, these beans are managed by Spring Annotations
* Spring Annotations, are the alternative way to define beans and their dependencies. This method is a Java-based configuration.
* Unlike the XML approach, Java-based configuration allows you to manage bean components programmatically. That is why Spring annotations were introduced.

**Defining Beans**

Two ways of defining beans

1. Using Stereotype Annotations

* Annotate your class with one of the stereotype annotations (@Component, @Service, @Repository, @Controller). These annotations inform Spring that the class should be managed as a bean.
* @SpringBootApplication is the starting point of any spring boot application

1. Explicit Bean Declaration in the Configuration Class
   * Create a configuration class and annotate it with @Configuration. This class will contain methods to define and configure beans. It is used for creating manual beans.

**Bean Lifecycle**

1. Bean Created
   1. The bean instance is created by invoking a static factory method (defining bean in the configuration class) or an instance factory method (@Component)
2. Dependency Injected
   1. After the creation of the bean, Spring sets the beans properties and dependencies, either through setter injection, constructor injection, or field injection.
3. Bean Initialized
   1. If a bean implements the initializingbean interface or define a custom initialization method annotated with @PostConstruct, Spring invokes the initialization method after the bean has been configured.
4. Bean is Used
   1. The bean is now fully initialized and ready to be used by the application
5. Bean destroyed
   1. Spring invokes the destruction method when the bean is no longer needed or when the application context is being shut down

**Bean Lifecycle Hooks**

1. The @PostConstruct annotation is used to mark a method that should be invoked immediately after a bean has been constructed and all of its dependencies have been injected.
2. The @PreDestroy is used to mark a method that should be invoked just before a bean is destroyed by the container. This method can perform any necessary cleanup or resource releasing tasks.

**Scope of Beans**

Beans are stored in heap memory

|  |  |
| --- | --- |
| Scope | Description |
| Singleton | (Default) Scopes a single bean definition to a single object instance for each Spring IoC container |
| Prototype | Scopes a single bean definition to any number of object instances |
| request | Scopes a single bean definition to the lifecycle of a single http request. That is, each http request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring Application context |
| WebSocket | Scopes a single bean definition to the lifecycle of a websocket. Only valid in the context of a web-aware Spring applicationContext |

**Dependency Injection**

### 🧃 Analogy: Making a Smoothie

Imagine you're a **Smoothie Maker** (like a blender), and your job is to make a smoothie. To do that, you need ingredients like bananas, strawberries, and milk.

Now, consider two ways of getting those ingredients:

### ❌ Without Dependency Injection:

You, the **Smoothie Maker**, go to the grocery store, pick the bananas, strawberries, and milk **yourself**, then come back and blend them.

* This means you're responsible for finding and choosing the ingredients.
* If the recipe changes (e.g., use almond milk instead of regular milk), **you** have to change how you shop.

This is tightly coupled — you're doing everything.

### ✅ With Dependency Injection:

Instead, someone **hands you** the bananas, strawberries, and milk directly — ready to blend.

* You don’t care where they came from or how they were chosen.
* You just do your job: **blend**.

This is loosely coupled — you focus only on blending, and someone else provides the needed parts (dependencies).

### In Code Terms:

* **You = Class**
* **Ingredients = Dependencies**
* **Someone handing them to you = Dependency Injection**

### 🔧 Why It's Useful:

* Easier to **change ingredients** (swap dependencies).
* Easier to **test** (you can hand in mock ingredients).
* Your class becomes **simpler and more focused**.
* It is used to achieve loose coupling between components in a software application
* IoC manage Dependency injection

**Benefits of Dependency Injection**

Loose Coupling: Components are decoupled from their dependencies, making them easier to maintain and test.

Flexible Configurations: Dependencies can be configured externally, allowing for easier customization and swapping of components.

Improved Testability: Components can be easily mocked or replaced during testing, allowing for more thorough and isolated unit tests.

A couple of ways to inject dependencies

1. Constructor Injection
   1. Dependencies are provided through a class constructor; you can make your fields private
2. Field Injection
   1. Dependencies are provided directly into the fields of a class using @Autowired

**Spring Boot vs Spring Framework**

* In spring Boot, we can use dependencies
* In Spring Boot, we have autoconfiguration
* In Spring Boot, we have Externalized configuration
* In Spring Boot, we have embedded tomcat, Jetty Servers
* In Spring Boot, we get built-in metrics and health checks

**Pom.xml**

* Maven us a popular build automation tool used in many Java projects. In a Spring Boot project, dependencies are specified in the pom.xml file. Maven then resolves these dependencies and includes them in the classpath.
* Starters like spring-boot-starter-parent include a ton of third-party libraries into the spring boot project by default. Its autoconfigurations use these dependencies to setup and preconfigure these libraries automatically.
* The spring-boot-dependencies pom.xml contains every 3rd party library (and version) that Spring Boot knows. These libraries are predefined in a dependenciesManagment section, so you do not need to specify the version numbers in your own project.

**What is Auto Configuration**

Autoconfiguration refers to the mechanism that automatically configures Spring applications based on the dependencies present on the classpath and other application-specific settings.

This feature simplifies the setup and development process, allowing developers to focus more on writing business logic rather than configuring the framework.

**How Autoconfiguration Works**

1. Classpath Scanning
   1. Spring Boot scans the classpath for the presence of certain libraries and classes. Based on what it finds, it applies corresponding configurations.
2. Configuration Classes
   1. Next, Spring Boot contains numerous autoconfiguration classes, each responsible for configuring a specific part of the application.
3. Conditional Beans
   1. Each autoconfiguration uses conditional checks to decide if it should be applied. These conditions include the presence of specific classes, the absence of user-defined beans, and specific property settings.

**Core Features of AutoConfiguration**

* @PropertySources Auto-registration
  + When you run the main method of your Spring Boot Application, Spring Boot will automatically register 17 of the PropertySources for you.
* META –
  + Every Spring Boot project has a dependency on the following library: org.springframework.boot:spring-boot-autoconfigure. It is a simple .jar file containing pretty much all of Spring Boot magic.
* Enhanced Conditional Suupport
  + Spring Boot comes with its own set of additional @Conditional annotations, which make developers lives easier.
  + @ConditionalOnBean(DataSource.class). The condition is ture only if the user specified a DataSource @Bean in a @COnfiguuration
  + @ConditionalOnClass(DataSource.class). The condition is true if the DataSouruce class is on the classpath.
  + @ConditionaslOnProperty(“my.property”). The condition is true if my.property is set.

**Spring FrameWork + AutoConfiguration = Spring Boot**

**Spring Boot Internal Flow**

1. **Initialization:** When we start our Spring Boot application, the main entry point is typically a class annotated with @SpringBootApplication (or its meta-annotations). This annotation combines several other annotations such as @Configurations, @EnableAutoConfiguration, and @ComponentScan.
2. **Spring Application**: It scans the classpath for components, configurations, and auto-configurations, and initializes the application context based on the detected classes and dependencies.
3. **Auto-Configuration:** Spring Boot auto-configures beans and components based on the classpath and detected dependencies. It uses conditional annotations (@ConditionalOnClass, @ConditionalOnBean, etc) to conditionally configure beans only if certain conditions are met.
4. **Externalized Configuration**: Spring Boot loads configuration properties from various sources, such as property files, YAML files, environment variables, and command-line arguments. It provides sensible default values for configuration properties and allows them to be easily overridden or customized.
5. **Embedded Web Server Initialization**: If the application is a web application, Spring Boot initializes the embedded web server (such as tomcat) based on the application’s dependencies and configurations. It configures the server with sensible defaults and starts it to listen for incoming requests.
6. **Application Startup**: Spring Boot invokes lifecycle callbacks such as @PostConstruct methods and initialization callbacks on beans as the application context is being initialized. Beans are instantiated, dependencies are injected, and any necessary initialization logic is executed.
7. Application Ready: Once the initialization process is complete, the application context is fully initialized and ready to handle requests. The embedded web server is up and running, and the application is ready to serve incoming HTTP requests.

**Maven**

Maven is a build automation tool and project management tool primarily used for Java Projects. It plays a crucial role in the development, build, and dependency management of Spring applications, including those using Spring Framework’s component scanning and annotation driven configuration.

**Maven Project and Dependency Management**

* Maven provides a standardized way to manage Java Projects by defining project structure, dependencies, and build configurations using a declarative XML-based format (pom.xml)
* Developers use Maven to specify project metadata, dependencies, plugins, repositories, and other project-related configurations.
* Spring Framework and its various modules (e.g., Spring core, Spring MVC, Spring Boot) are managed as dependencies in Maven projects. Developers specify the Spring dependencies in the pom.xml, and Maven handles the rest.

**Maven Build Automation**

* Maven automates the build process, including compilation, testing, packaging, and deployment, using predefined build lifecycle phases (e.g., clean, compile, test, package, install).
* Maven facilitates the building and packaging of Spring applications into deployable artifacts (e.g., JAR files, WAR files) for deployment in production environments.

## 🧠 Annotations Explained (with the Toy Car Analogy)

### 1. @SpringBootApplication

🛠 **Main switch to start the toy car project**

Think of this as **pressing the "ON" button on your toy car kit**. It tells Spring:

“Start the engine, connect all the parts, and get the car running.”

It combines 3 things under the hood:

* @Configuration: "Here's how to build stuff."
* @EnableAutoConfiguration: "Auto-assemble common parts."
* @ComponentScan: "Look around and find parts (components, services, etc)."

### 2. @Service

⚙️ **Label for the Engine**

This tells Spring:

“This class is doing a core job — like powering the car.”

You usually use this for **business logic**, like calculating speed, distance, or controlling movement.

### 3. @Component

📦 **General-purpose labeled part**  
Imagine it as any other part like a **GPS sensor or horn**.

Spring sees this label and says:

“Okay, I’ll manage this part and plug it in where needed.”

It’s a **generic version** of @Service, @Repository, etc.

### 4. @Configuration

📘 **The toy car manual**

This says:

“Hey Spring, I’ve written down how to make some parts. Use these instructions.”

You use this to define custom beans (like your own type of Battery or Engine).

### 5. @Autowired

🔌 **Battery slot that says "Insert here automatically"**

You don't place the battery yourself. You just write:

java

CopyEdit

@Autowired

Battery battery;

Spring sees it and says:

“Ah! He needs a Battery. I’ll put one here from the parts I already created.”

It’s **auto-wiring** dependencies (connecting pieces together).

### 6. @Bean

🧪 **Custom part from the manual**

You write a function that says how to **build a specific part**, like this:

java

CopyEdit

@Bean

public Battery battery() {

return new Battery("Rechargeable");

}

Spring reads it and creates that part when needed.

### 7. @Scope

📦 **How many parts should I build?**

Used with @Bean or @Component to tell Spring:

“Do you want a new part every time, or just one shared part?”

Example:

* @Scope("singleton"): Only one battery for all cars.
* @Scope("prototype"): A new battery for every car.

### 8. @PostConstruct

🔧 **Run this right after building the part**

Once Spring finishes making the part, you can run some setup logic (like testing the wheels after installing them).

java

CopyEdit

@PostConstruct

public void init() {

System.out.println("Wheels are spinning!");

}

### 9. @PreDestroy

🧹 **Cleanup before trashing the part**

Before Spring removes or shuts down the part, it runs cleanup — like turning off the lights or unplugging the battery.

java

CopyEdit

@PreDestroy

public void cleanup() {

System.out.println("Car is shutting down...");

}

### 10. @ConditionalOnProperty

🕹 **Only build the part if the switch is ON**

Used when you want to include a part **only if some setting is true** in your config file:

java

CopyEdit

@ConditionalOnProperty(name = "car.gps.enabled", havingValue = "true")

This says:

“Only add GPS if it’s enabled in the settings.”

## ✅ Summary: Toy Car Parts & Spring

| **Annotation** | **What It Does** |
| --- | --- |
| @SpringBootApplication | Starts the whole toy car factory (main switch) |
| @Service | Marks important business logic (engine) |
| @Component | Marks general-purpose parts (GPS, horn) |
| @Configuration | The manual for making custom parts |
| @Autowired | Auto-attach the needed parts (like putting battery in slot) |
| @Bean | Define a custom part to be created (battery instructions) |
| @Scope | Controls how many times to create that part (single or multiple batteries) |
| @PostConstruct | Run something **right after** part is created (test the wheels) |
| @PreDestroy | Run something **before** part is removed (shut it down) |
| @ConditionalOnProperty | Only add part **if** a setting is true (like GPS switch ON) |