**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 Soring 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.