Understanding Core Concepts in Spring

SPRING FRAMEWORK LEARNINGS – KEY LEARNINGS (PART 2)

Spring Container

- The Spring Container is the core of the Spring Framework.
- It is responsible for managing the lifecycle of Spring beans: their creation, configuration, and destruction.
- It performs Dependency Injection to wire dependencies.
- Two main implementations: BeanFactory and ApplicationContext.
- Without Spring → Developers create objects manually like: Car car = new Car();
- But with Spring → No need to do this manually.
- Spring Container does this automatically for you based on configuration.

Spring Container

- **Real-life Example:**
- Think of Spring Container like a Coffee Machine.
- You press a button → It automatically creates coffee for you.
- You don't worry about coffee beans, milk, water (dependencies).
- It manages everything internally.

```
@Component  // Spring Bean
class Coffee {
 String coffeeType = "Latte";
 String milk = "Yes";
 String sugar = "Low";}
// Spring Container will do this for you
class Person {
   @Autowired
   Coffee c;  // No need to create manually
 public void drinkCoffee() {
      System.out.println("Coffee Ready: " + c.coffeeType);
   }  }
```

@Component and @ComponentScan

- @Component

  → Marks any Java Class as *Spring Bean*
  → Means → *Spring will automatically create object of this class*

- Example: Web Series uploaded on Netflix
- *@ComponentScan*

  → Tells Spring

  → *Where to search for @Component classes*

  → By default searches for component classes in the current package

→ @ComponentScan("path of the package we want to search")

- Example: Netflix scans folders to find new Web Series

BeanFactory vs ApplicationContext in Spring

Features	Bean Factory	Application Context
Bean Creation	Lazy Loading (creates bean when asked)	Eager Loading (creates beans at startup)
Features	Basic Only	Advanced (Event handling, i18n, profiles, etc.)
Used In	Small apps, memory-sensitive apps	Enterprise/Real-world Spring projects
Spring Boot	Not used	Mostly used by default

POJO vs Java Beans vs Spring Beans

- **POJO:** Very simple normal Java class. No special rules or restrictions.
- **Java Bean** = POJO + Some Rules
- **Rules of Java Bean:**
- Must have *private* variables
- Must have *public* getters & setters
- Must have *no-arg constructor*
- Should be *Serializable* (Optional but recommended)
- **Spring Bean:** *POJO or Java Bean + Managed by Spring Container*
- Any object that is managed by *Spring Container* is called a *Spring Bean*.
- Spring Beans can be Java Beans, but not all Java Beans are Spring Beans.

IoC (Inversion of Control)

- Inversion of Control (IoC) means delegating object creation to a container.
- i.e. Control of creating object is inverted.
- Spring Container controls it & provides ready object.
- IoC (Inversion of Control) = *Give control to Spring*
- Normally:
  Class A → Creates Class B

  With IoC:
  Spring Container → Creates Class A + Class B
  Spring Container → Injects B inside A

Dependency Injection

- Dependency Injection (DI) is a way to implement IoC.
- Spring uses DI to inject dependencies into objects instead of creating them manually.

Shortcut Example

Without DI :	**With DI:**
You make Pizza yourself	Pizza delivery boy brings ready Pizza
Tightly Coupled	Loosely Coupled

- Types of DI: Constructor-based (preferred), Setter-based, Field-based.

Types of Dependency Injection in Spring

- Constructor Injection: Injects dependencies via constructor – ensures immutability and is test-friendly.
- Example:

```
@Component
class Person {
    Pizza pizza;
    @Autowired
    Person(Pizza pizza) {   this.pizza = pizza;  } }
```

- Setter Injection: Injects via setter methods – good when dependencies are optional.
- Example:

```
@Component
class Person {
    Pizza pizza;
    @Autowired
    public void setPizza(Pizza pizza) {  this.pizza = pizza; } }
```

- Field Injection: Uses @Autowired on fields – not recommended due to poor testability.
- Example:

```
@Component
class Person {
    @Autowired
      Pizza pizza;  }
```

Autowiring and @Autowired

- The process of automatically injecting dependent objects by Spring is called → *Autowiring*
- @Autowired tells Spring →
  Give me this object automatically from your Container, I don't want to create it manually.
- Example:

```
@Component
    class Pizza { String type = "Cheese Burst"; }
@Component
    class Order {
        @Autowired
         Pizza pizza;  // Auto Injected by Spring }
```

No need to write:

```
      pizza = new Pizza();
```

- @Autowired is used to auto-wire beans by type.
- It can be used on constructors, setters, and fields.
- Spring resolves the dependencies automatically based on type (and name, if ambiguity arises).

Handling Multiple Matching Beans

- When multiple beans of the same type exist, Spring may not know which one to inject.
- Use @Primary to mark one bean as the default.
- Use @Qualifier("beanName") to explicitly specify the bean to inject.
- @Qualifier overrides @Primary when both are used.

Example:

Annotation	Meaning	Real Life Example
@Primary	Default Bean	Preferred Pizza Shop
@Qualifier	Specific Bean	Exact Shop You Ordered From

Summary

- Spring Container manages bean lifecycle and dependencies.
- Use constructor injection for better testability and design.
- Handle multiple beans with @Primary and @Qualifier.
- Use @Component for class-level beans and @Bean for external classes.
- ApplicationContext is the go-to container for modern Spring applications.