Spring Boot 3 - Inversion of Control and Dependency

What is Inversion of Control?

- **Inversion of Control (IoC)**
  - The approach of outsourcing the construction and management of objects
- **Spring Container**
  - **Primary Functions**
    - Create and manage objects **(Inversion of Control)**
    - Inject object dependencies **(Dependency Injection)**
- **Configuring Spring Container**
  - **XML** configuration file → Legacy
  - **Java** Annotations → Modern
  - **Java** Source Code → Modern

Defining Dependency Injection

- The **Dependency Inversion Principle**
  - The client **delegates** to another object the **responsibility** of **providing** its **dependencies**
- **Injection Types**
  - There are **multiple types** of **injection** with **Spring**
  - Two **recommended types** of **injection**
    - **Constructor** Injection
    - **Setter** Injection
- **Injection Types** - Which to Use?
  - **Constructor Injection**
    - Use this when you have **required dependencies**
    - Generally **recommended** by the **spring.io** development team as first choice
  - **Setter Injection**

        - Use this when you have **optional dependencies**
        - If a **dependency isn't provided**, your app can **provide reasonable logic**
- What is **Spring Autowiring?**
    - **Spring** can use **autowiring** for **dependency injection**
    - **Spring** looks for a class that matches
        - **Matches By Type:** class or interface
    - **Spring injects automatically**
- Development Process → **Constructor Injection**
    - Define the **dependency interface** and **class**
    - Create Demo REST **Controller**
    - Create a **constructor** in your **class** for **injections**
    - Add @GetMapping for **/dailyworkout**
- **@Component** annotation
    - **@Component** marks the class as a **Spring Bean**
        - A **Spring Bean** is just a regular **Java class** that is **managed by Spring**
    - **@Component** also makes the **bean** available for **dependency injection**

IDE Warning - No Usages

- **Spring Framework** is **dynamic**
- The **IDE** may be **unable** to **determine** if a **class/method** is used at **runtime**

Constructor Injection - Behind the Scenes

- **How Spring Processes Your Application**
    - We have an **interface**, a **class**, and a **controller**
    - **Spring instantiates** the **class** and performs **instructor injection** with the **controller**
- **Spring** for e**nterprise applications**
    - **Spring** is targeted for **enterprise**, **real-time / real-world applications**

    - **Spring** provides features such as:
        - **Database access** and **transactions**
        - **REST APIs** and **Web MVC**
        - **Security**
        - etc.

Component Scanning

- **Scanning for Component Classes**
    - **Spring** will **scan** your **Java classes** for **special annotations**
        - **@Component**, etc.
    - **Automatically register** the **beans** in the **Spring** container
- **Annotations**
    - **@SpringBootApplication** is composed of the following annotations:

Annotation	Description
@EnableAutoConfiguration	Enables Spring Boot's auto-configuration support
@ComponentScan	Enables component scanning of current package; also recursively scans sub-packages
@Configuration	Able to register extra beans with @Bean or import other configuration classes

- More on **Component Scanning**
    - By **default**, **Spring Boot** starts **component scanning**
        - From the **same package** as your main **Spring Boot** application
        - Also **scans sub-packages recursively**
    - This **implicitly defines** a **base search package**
        - Allows you to **leverage default component scanning**
        - **No need** to **explicitly reference** the **base package name**
    - To **scan packages outside** of the **base**, you can **explicitly list** them in the main file

Setter Injection

- **Setter Injection** is when we **inject dependencies** by calling **setter methods** on your **class**
- **Autowiring Example**
  - **Injecting** a **Coach implementation**
  - **Spring** will scan for **@Components**
  - **Inject** any class **implementing** the **Coach** interface
- **Development Process**
  - Create **setter methods** in your **class** for **injections**
  - **Configure** the **dependency injection** with the **@Autowired** annotation
- **Injection Types - Which to use?**
  - **Constructor Injection**
    - Use this when you have **required dependencies**
    - Generally **recommended** by the **spring.io** development team as the first choice
  - **Setter Injection**
    - Use this when you have **optional dependencies**
    - If a **dependency isn't provided**, your **app** can provide **reasonable default logic**

Field Injection

- **Not recommended** by the **spring.io** development team
- **Was popular** on **Spring** projects
- Makes the code **harder** to **unit test**
- **Inject dependencies** by **setting field values** on your **class** directly
  - Including **private fields**
  - Accomplished using **Java Reflection**
- **Configure** the **dependency injection** with **@Autowired**

Qualifiers

- **Autowiring**
  - **Injecting** a **Coach implementation**

    - **Spring** will scan **@Component**s for anyone **implementing Coach**
    - How does it pick which **class** to inject if **multiple exist**?
- Resolve this problem using **@Qualifier**
    - Pass in the specific **Bean ID** → **Class name** in **camel-case**
    - Works for both **Constructor/Setter Injection**

Primary

- Instead of **specifying** a **Coach** with **@Qualifier**, we let **Spring** determine the **primary Coach**
- **No** longer **need** to use the **@Primary** decorator
- We inject the **@Primary** decorator to the **primary class**
- There can only be **one** class with **@Primary** → Errors with two or more
- **Mixing @Priority** and **@Qualifier**
    - Possible to mix, but **@Qualifier** has **priority**
- Which to use: **@Primary** or **@Qualifier**?
    - **@Primary** leaves it up to the **implementation class**
        - Could have the issue of **multiple @Primary** classes leading to an **error**
    - **@Qualifier** allows you to be very **specific** on which **bean** you want
        - Better since you can be **more specific** and has **higher priority**

Lazy Initialization

- **Initialization**
    - By default, all **beans** are **initialized** at startup
        - **@Component**, etc.
    - **Spring creates** an **instance** of **each bean** and makes them **available**
- **Lazy Initialization**
    - Instead of creating all beans up front, we can specify **lazy initialization**
    - A **bean** will only be **initialized** in the following cases
        - It's needed for **dependency injection**
        - It's **explicitly requested**

    - Add the **@Lazy** annotation to a given class
- **Lazy Configuration - Global Configuration**
    - **spring.main.lazy-initialization=true**
    - **All beans** are **lazy** → **none** are **created** until **needed**
    - **Disabled** by default
- **Advantages**
    - Only **create objects** as **needed**
    - May help with **faster startup time** if you have a large number of components
- **Disadvantages**
    - Web-related components (like **@RestController**) **won't be created until requested**
    - **May not discover configuration issues** until too late
    - Need to ensure you **have enough memory** for all beans once created

Bean Scopes

- **Scope** refers to the **bean's lifecycle**
    - How **long** does the bean live?
    - How many **instances** are created?
    - How is the bean **shared**?
- Default Scope → **Singleton**
    - **Spring Container** creates only **one instance** of the **bean**
    - It's **cached** in memory
    - All **dependency injections** for the **bean reference** the **same bean**
- Additional **Spring Bean Scopes**

Scope	Description
Singleton	Creates a single shared instance of the bean; default scope
Prototype	Creates a new bean instance for each container request

Request	Scoped to an HTTP web request; only used for web apps
Session	Scoped to an HTTP web session; only used for web apps
Global-Session	Scoped to a global HTTP web session; only used for web apps

Bean Lifecycle Methods

- **Bean Lifecycle**
  - **Container started** → **Bean instantiated** → **Dependencies injected** → **Internal Spring Processing** → Your **Custom Initialization Method**
  - **Bean** is **ready to use**
  - **Container** is **shut down** → Your **Custom Destroy Method** → **Stop**
- **Bean Lifecycle Methods / Hooks**
  - You can add **custom code** during **bean initialization**
    - Calling **custom business logic methods**
    - Setting up **handles** to resources (DB, Sockets, File, etc.)
  - You can add **custom code** during **bean destruction**
    - Calling **custom business logic** method
    - Clean up **handles** to **resources**
- **Development Process**
  - Define your **methods** for **init** and **destroy**
  - Add annotations: **@PostConstruct** and **@PreDestroy**

Special Note About Prototype Scopes

- **Spring** doesn't call the **destroy method** on **prototype-scoped beans**

Java Config Bean

- **Development Process**
  - Create **@Configuration** class
  - Define **@Bean method** to **configure** the **bean**
  - **Inject** the **bean** into the **controller**

- Use case for **@Bean**
  - Make an **existing third-party class available** to **Spring Framework**
  - You may **not have access** to the **source code** of the **third-party class**
  - However, you would like to **use** the **third-party class** as a **Spring bean**