DAY 3-4: INTRODUCTION TO SPRING BOOT

SPRING BOOT 3.4.2 FRAMEWORK FUNDAMENTALS

Definition:

Spring Boot is a framework built on top of Spring that simplifies the creation of production-ready, stand-alone Spring applications with minimal configuration.

Key Features:

1. **Auto-Configuration:** Automatically configures Spring components based on dependencies in the classpath.
2. **Standalone:** Embedded server (Tomcat, Jetty, or Undertow) – no need for external server setup.
3. **Opinionated Defaults:** Provides pre-configured settings to reduce boilerplate code.
4. **Production-ready:** Includes metrics, health checks, and monitoring support.
5. **Easy dependency management:** Works with Maven or Gradle.

Example: Minimal Spring Boot Application

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

@SpringBootApplication  // Enables auto-configuration, component scan
public class DigitalWalletApplication {
   public static void main(String[] args) {
      SpringApplication.run(DigitalWalletApplication.class, args);
   }
}
```

Explanation:

- @SpringBootApplication is a combination of:
    - @Configuration – allows defining beans.
    - @EnableAutoConfiguration – triggers auto-configuration.

- @ComponentScan – scans for Spring components (controllers, services, repositories).

Why use Spring Boot?

- Reduces configuration effort.
- Quickly starts new projects.
- Integrates easily with databases, messaging systems, and REST APIs.

DEPENDENCY INJECTION AND INVERSION OF CONTROL PRINCIPLES

Inversion of Control (IoC) is a design principle where an external framework or container manages the flow of control, including the creation and management of objects and their dependencies, rather than the application code itself.

Dependency Injection (DI) is a concrete design pattern that implements IoC by providing an object's dependencies (other objects it needs) to it from the outside, rather than the object creating them itself. Together, these concepts promote loose coupling, improved testability, and enhanced maintainability in software.

Inversion of Control (IoC)

- **The Principle**: The core idea is to "invert" the traditional flow of control. Instead of a class directly creating its dependencies or controlling the overall application flow, a separate entity (like a framework or container) takes over this responsibility.
- **How it Works**: An external source dictates the execution flow and provides the necessary components to your application's classes.
- **Benefits**:
    - Increased Modularity: Components become more independent and reusable.
    - Improved Testability: Easier to isolate components and test them with mock dependencies.
    - Flexible Implementations: Allows for easy switching between different implementations of a dependency without changing the core class.

Dependency Injection (DI)

The Pattern: DI is a way to achieve IoC. Instead of a class requesting its dependencies, the dependencies are "injected" into the class from an external source.

- **How it Works: Dependencies are typically provided to a class through:**
    - Constructor Injection: Dependencies are passed as arguments to the class's constructor.
    - Setter Injection: Dependencies are set via setter methods.
- **Benefits:**
    - Decoupling: Classes are not tied to the concrete implementations of their dependencies, leading to more loosely coupled code.
    - Reusability: Components become more reusable because their dependencies are not hardcoded.
    - Simpler Maintenance: Changes in one component are less likely to affect others.

Relationship Between IoC and DI

- IoC is a broad design principle that suggests controlling the flow of a program should be handled by an external source, not by the program itself.
- DI is a specific pattern to implement the IoC principle. By injecting dependencies from an external source, you are inverting the control of dependency creation **from** the class to the injector.

SPRING BOOT AUTO-CONFIGURATION MECHANISM

Definition:
Spring Boot automatically configures beans and settings based on the dependencies present in the project.

- Reduces manual configuration.
- Lets you focus on business logic instead of setup.

How It Works:

1. Spring Boot scans your classpath for libraries (starters).
2. Determines what you need (e.g., web server, database, messaging).
3. Automatically creates default beans for you.
4. You can override defaults by defining your own beans.

Example: Web Application

Dependency in build.gradle:

implementation 'org.springframework.boot:spring-boot-starter-web'

- Spring Boot auto-configures:
    - Embedded Tomcat server
    - Spring MVC (@Controller, @RestController)
    - JSON serialization (Jackson)

Minimal Controller Example:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;


@RestController
public class HelloController {
   @GetMapping("/hello")
   public String sayHello() {
      return "Hello, Spring Boot!";
   }
}
```

- No server configuration needed.
- Run the app → http://localhost:8080/hello works out-of-the-box.

SPRING BEAN LIFECYCLE & SCOPES

Definition:

- A Bean is an object managed by the Spring container.
- Spring controls its creation, initialization, and destruction.

Bean Lifecycle Steps:

1. Instantiation – Spring creates the bean instance.
2. Dependency Injection – Spring injects required dependencies.
3. Initialization – Bean initialization methods (@PostConstruct or afterPropertiesSet()) run.
4. Ready to Use – Bean is fully initialized and available for use.
5. Destruction – Bean is destroyed when the application context closes (@PreDestroy or destroy()).

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 ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

websocket	Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

CONFIGURATION MANAGEMENT USING APPLICATION.PROPERTIES & YAML

Definition:

Spring Boot allows you to **externalize configuration** so that your app behavior can change without modifying the code.

- Files: application.properties (key-value) or application.yml (hierarchical).
- Used for: server settings, database configs, profiles, logging, etc.

1. Using application.properties

Example:

```
server.port=8081
spring.profiles.active=dev
spring.datasource.url=jdbc:mysql://localhost:3306/wallet
spring.datasource.username=root
spring.datasource.password=root
logging.level.org.springframework=INFO
```

- **server.port** → change HTTP port.
- **spring.profiles.active** → select environment (dev/test/prod).
- **spring.datasource** → database configuration.
- **logging.level** → control logging verbosity.

2.Using application.yml

Example:

```
server:
  port: 8081
```

```
spring:
  profiles:
    active: dev
  datasource:
    url: jdbc:mysql://localhost:3306/wallet
    username: root
    password: root

logging:
  level:
    org.springframework: INFO
```

- YAML is **structured**, easier for hierarchical configs.

3. Profile-based Configuration

- You can create environment-specific files:
    - application-dev.properties
    - application-prod.yml
- Spring Boot automatically loads the correct file based on spring.profiles.active.

GRADLE PROJECT STRUCTURE & DEPENDENCY MANAGEMENT

Definition:

Gradle is a **build automation tool** used to compile code, manage dependencies, run tests, and build Spring Boot applications.

1.Gradle Project Structure

```
project/
 ├─ build.gradle          # Project dependencies & build config
 ├─ settings.gradle       # Project name & module settings
 ├─ src/
 │   ├─ main/
 │   │   ├─ java/         # Application source code
 │   │   └─ resources/    # Configuration files (application.properties/yml)
 │   └─ test/
 │       ├─ java/         # Test source code
 │       └─ resources/    # Test resources
 └─ gradle/               # Gradle wrapper files
```

2. build.gradle (Dependencies & Plugins)

Example:

```
plugins {
    id 'org.springframework.boot' version '3.4.2'
    id 'io.spring.dependency-management' version '1.1.2'
    id 'java'
}

group = 'org.transactions'
version = '1.0.0'
sourceCompatibility = '17'

repositories {
```

```
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    runtimeOnly 'com.mysql:mysql-connector-j'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- **plugins:** Add Spring Boot and dependency management capabilities.
- **repositories:** Source of external libraries (Maven Central).
- **dependencies:**
    - implementation → compile-time + runtime dependencies
    - runtimeOnly → only needed at runtime
    - testImplementation → only for testing

3. Gradle Tasks

- ./gradlew build → build the project
- ./gradlew bootRun → run Spring Boot app
- ./gradlew test → run tests
- ./gradlew clean → clean build files