Spring Framework Guide

Spring Framework

Product Controller will accept requests and send them to the service layer, where we write business logic. For database interactions, we use the DAO (Data Access Object) layer, which communicates with the database, retrieves data, and passes it back to the service layer, which then forwards it to the controller.

Product DAO contains methods for interacting with the database.

Dependency Injection (DI) and Inversion of Control (IOC):
- Object injection from one place to another is managed by the IOC Container.
- With Spring JDBC, we can connect to databases easily through its API.

Spring ORM

- **Spring Web Module** allows us to create REST APIs for web applications, including image uploads.
- **Test Module** supports JUnit and TestNG for application testing.

Inside Spring Framework

- **IOC Container**:
  - Creates objects, holds them in memory, and injects them when required.
  - Requires bean and configuration information.
  - Reads configuration files, creates bean objects, and injects dependencies.

- **Application Context** is an interface representing IOC. It includes classes such as `ClasspathXmlApplicationContext`, `AnnotationConfigApplicationContext`, etc.

- The class provided to the IOC container is called a **Bean**.
- `@Bean` eliminates the need for XML configuration files.
- **Constructor & Setter Injection**: Used to set values in objects.

Spring Container provides two lifecycle methods for beans:

- **init**: Loads configuration files, connects to the database, and uploads web services.

- **destroy**: Cleans up resources.

AutoWiring in XML

- `byName`

- `byType`

- `byConstructor`

Spring Boot

Request Flow

1. The first request goes to the **Dispatcher Servlet**.

2. Based on mappings, it invokes the respective controller.

Why Spring Boot?

- Traditional Spring requires manually adding dependencies like Spring Web MVC, Jakarta Servlet, and configuring a Dispatcher Servlet.

- Spring Boot simplifies setup and includes an embedded Tomcat Server.

Spring Boot Components

- **Spring Boot Starter Web**

- `@EnableAutoConfiguration`: Enables the server and dispatcher servlet.

- `@ComponentScan`: Scans all classes in the package.

- `@SpringBootApplication`: A combination of both.

- `@Component`: Marks a class as a Spring Bean, allowing the IOC Container to manage it.

- `@Bean`: Used to create and manage beans manually.

Application Context

- Acts as the container.

- The `run` method is used to create the container and manage beans.

JAR File

- A collection of `.class` files.

REST API

REST API enables communication between two applications using JSON/XML data.

- **Controller** handles incoming requests and sends responses.
- **Annotations**:
  - `@PathVariable`: Binds URL parameters to method parameters.
  - `@RequestParam`: Retrieves values from form data.
  - `@Autowired`: Injects dependencies.
  - `@Qualifier`: Helps resolve bean conflicts.
  - `@Primary`: Alternative to `@Qualifier`.
  - `@Entity`: Creates database tables.
  - `@RequestBody`: Accepts JSON data.
  - `@Configuration`: Replaces XML configuration.

Spring Data JPA & Hibernate

- **JDBC** requires writing SQL queries and handling exceptions manually.
- **Hibernate** automates table creation and provides Spring Data JPA.
- **JpaRepository** includes methods like:
  - `save()`, `update()`, `findAll()`, `findById()`, `delete()`.

Spring Boot Application Flow

1. **application.properties** - Provides database connection.
2. **Entity Layer** - Defines fields to store data in the table.
3. **Repository Layer** - Extends `JpaRepository`.
4. **Service Layer** - Uses repository methods for business logic.
5. **Controller Layer** - Calls service methods and handles HTTP requests.

DTO (Data Transfer Object)

- Used instead of returning entities directly from the service layer for security and data abstraction.
- Creates separate DTO response classes used in the service and controller layers.

Microservices

- A **collection of REST APIs** that work independently but communicate with each other.