Introduction To Spring Boot Web MVC, MVC architecture.

Different types of APIS

1. REST APIs
2. SOAP APIs
3. Web sockets

REST APIs

REST (Representational State Transfer) APIs (Application Programming Interfaces)) are a set of rules and conventions for building and interacting with web services. REST are stateless, each request is **independent and complete**.

* GET /users: Retrieve a list of all users.
* GET /users/{id}: Retrieve a specific user by ID
* POST /users: Create a new user
* PUT /users/{id}: Update an existing user by ID
* PATCH /users/{id}: Partially update an existing user by ID
* DELETE /users/{id}: Delete a user by ID

Spring-boot-starter-web

REST APIs are used in spring boot through spring-boot-starter-web dependency. It contains the following child dependencies.

* Spring-boot-starter:
  + contain all the autoconfiguration and IoC
* Jackson:
  + Handles serialization and deserialization of JSON to/from Java objects automatically, used in REST APIs.
* Spring-core:
  + Provides the core functionalities of the Spring Framework such as dependency injection, utility classes, and bean lifecycle management.
* Spring-mvc (Model. View, Controller):
  + Enables building RESTful web applications; includes support for controllers, request mapping, view resolvers, and more.
* Spring-boot-starter-tomcat:
  + embedded server

Why use MVC Architecture:

* Separation of Concerns
* Reusability
* Scalability
* Testability

Presentation Layer

* The first contact between the client and the Spring boot application with the help of the Http request
* It talks to the Spring boot embedded servers
* It also provides output to the client
* A **controller** is a class or function that **handles incoming requests** from the client (like a browser or app), **processes them**, and **sends back a response**.
* Controller is the Presentation layer, also the DTO

**What are DTOs?**

* **DTO** stands for **Data Transfer Object**.
* A DTO is a simple Java object used to **transfer data** between layers (e.g., controller → service → database) without exposing internal details like entity structures or business logic.

Annotated Controller

* Spring MVC provides an annotated-based programming model where @Controller and @RestController components use annotations to express request mappings, request input, exception handling and more.
* The @RestController annotation is a shorthand for @Controller and @ResponseBody, meaning all methods in the controller will return JSON/XML directly to the response body, where JSON is in Key-value pairs and XML is tag based.

Request Mapping

* We can use the @Request Mapping annotation to map requests to controllers’ methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types.

There are also HTTP method specific shortcut variants of @RequestMapping

* @GetMapping – to get the data
* @PostMapping – to post the data. To post a data you need a client such a postman
* @PutMapping – update the data
* @DeleteMapping – delete the data
* @PatchMapping – Patch the data (update a little part of your data)

Dynamic URL path (how to get data)

* @PathVariable
  + /employee/123
  + Use path variables when the parameter is an essential part of the URL path that identifies a resource.
* @RequestParam
  + /employees?id=123
  + Use Query parameters when the parameter is optional and used for filtering, sorting or other modifications to the request

RequestBody

* @RequestBody is used to bind the HTTP request body to a Java Object. When client (postman) sends data in the body of a request (JSON, XML) @RequestBody maps this data to a Java object.

Persistence Layer & JPA

JPA -> JPA Provider -> API JDBC -> Driver connector\J(MYSQL), Driver PostreSQL (PostgreSQL)

ORM

**ORM** stands for **Object-Relational Mapping**. It’s a technique that lets you interact with a **relational database** (like MySQL, PostgreSQL, or Oracle) using **objects in your programming language** instead of writing raw SQL queries.

**🔧 What does an ORM do?**

At its core, an ORM:

* **Maps database tables to classes**
* **Maps table rows to objects**
* **Maps columns to attributes/properties**

**✅ Example (Java + Hibernate)**

Assume you have a table:

CREATE TABLE users (

id INT PRIMARY KEY,

name VARCHAR(100),

email VARCHAR(100)

);

With an ORM like Hibernate, you can represent this as a Java class:

@Entity

@Table(name = "users")

public class User {

@Id

private int id;

private String name;

private String email;

// Getters and setters

}

Then, instead of writing SQL like:

sql

CopyEdit

SELECT \* FROM users WHERE id = 1;

You just write:

java

CopyEdit

User user = entityManager.find(User.class, 1);

**🚀 Benefits of Using ORM**

* **No need to write raw SQL** (less boilerplate)
* **Automatic CRUD operations**
* **Database abstraction** (easily switch databases)
* **Helps prevent SQL injection** (when used properly)
* **Object-oriented approach** aligns with application code

**⚠️ Downsides**

* Can be **less efficient** than optimized SQL in complex cases
* Learning curve
* Sometimes **hard to debug** performance issues

**Popular ORMs**

* **Java**: Hibernate, JPA (Java Persistence API)
* **Python**: SQLAlchemy, Django ORM
* **JavaScript**: Sequelize (Node.js), TypeORM
* **C#**: Entity Framework

JPQL

**PQL** stands for **Java Persistence Query Language**. It's a query language used in **Java Persistence API (JPA)** to interact with the database **using the object model**, not directly with tables and columns.

### 🧠 What Makes JPQL Special?

* **Object-oriented**: You query **entities and their properties**, not tables and columns.
* It’s similar to SQL, but it works on **Java classes**, not the relational schema.

### ✅ JPQL vs SQL

| **Feature** | **JPQL** | **SQL** |
| --- | --- | --- |
| Based on | Java entity classes | Database tables |
| Uses | Class names and fields | Table names and columns |
| Returns | Entity objects | Rows/columns |
| Example Entity | User class | users table |

### 🔍 Example

#### Suppose you have this entity:

java

CopyEdit

@Entity

public class User {

@Id

private Long id;

private String name;

private String email;

}

#### A JPQL query would look like:

java

CopyEdit

SELECT u FROM User u WHERE u.email = :email

* User is the Java class.
* u.email is the field, not the column.
* :email is a named parameter.

### 🧑‍💻 Code usage with EntityManager:

java

CopyEdit

TypedQuery<User> query = entityManager.createQuery(

"SELECT u FROM User u WHERE u.email = :email", User.class);

query.setParameter("email", "test@example.com");

User result = query.getSingleResult();

### 📌 When to Use JPQL

* For complex queries that can't be handled by simple repository methods (e.g., findByEmail)
* When you want database portability and clean object-based querying
* To write dynamic or custom queries in Spring Data JPA

H2 database is in memory database

The relationship between an **Entity** and a **Repository** in Java using **Spring Data JPA** is a **structured connection** where:

* The **Entity** represents a **table** in your database.
* The **Repository** provides **methods to interact with that entity**, like saving, updating, deleting, or querying records—without writing SQL.

**Spring Data JPA** is a part of the larger **Spring Data** project, and it simplifies the use of **Java Persistence API (JPA)** in Spring-based applications.

**🔍 What is JPA?**

JPA is a **Java specification** for managing relational data in Java applications. It lets you map Java classes to database tables and perform database operations using object-oriented code.

**🌱 What Does Spring Data JPA Do?**

Spring Data JPA builds on top of JPA and provides **additional features** and **boilerplate reduction**.

It:

* Eliminates the need to write most **DAO (Data Access Object)** code.
* Uses interfaces instead of implementing classes for CRUD operations.
* Supports powerful query generation from method names.
* Integrates well with **Spring Boot** and **Hibernate** (a common JPA provider).

**✅ Example**

java

CopyEdit

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

}

java

CopyEdit

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByName(String name);

}

In this example:

* No need to write SQL or implementation.
* JpaRepository gives you methods like save(), findAll(), deleteById() automatically.
* The method findByName(String name) works without defining a query — Spring builds it from the method name.

**💡 Benefits**

* Rapid development with less code.
* Easy pagination and sorting.
* Query derivation and custom queries using @Query.
* Integration with Spring’s ecosystem.

Entity annotation

In Java (specifically in JPA – Java Persistence API), the @Entity annotation is used to mark a class as a **JPA entity**, meaning it represents a table in a relational database.

**🔹 What is @Entity?**

* It tells JPA (and tools like Hibernate) that the class should be mapped to a **database table**.
* Each instance of that class represents a **row** in the table.

**🧠 Simple Analogy:**

Think of a class marked with @Entity as a **blueprint for a database table row**. When you create an object of that class, it’s like inserting a new row into the table.

**✅ Example:**

java

CopyEdit

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

@Entity

public class Employee {

@Id

private Long id;

private String name;

private String department;

// Getters and setters

}

In the example above:

* @Entity tells JPA to map the Employee class to a database table called employee.
* @Id marks the id field as the **primary key**.

**📝 Key Notes:**

* The class **must have a no-arg constructor**.
* The class **must be public and not final**.
* It **must have a primary key**, marked with @Id.
* By default, the table name is the class name. You can customize it with @Table(name = "custom\_name").

**What is a JPA Repository Interface?**

In **Spring Data JPA**, a **JPA repository interface** is an abstraction layer that allows you to **perform CRUD operations** (Create, Read, Update, Delete) on a database **without writing SQL or JPQL**.

It **extends predefined interfaces** from Spring Data to give you **built-in methods** like save(), findById(), findAll(), deleteById(), etc.

**✅ Basic Example:**

java

CopyEdit

import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// You can also define custom query methods here

List<Employee> findByDepartment(String department);

}

Here:

* Employee is your @Entity class.
* Long is the type of the primary key (id).
* JpaRepository gives you powerful built-in database methods.
* findByDepartment is a **custom query method**: Spring auto-generates the query based on the method name.

**🧠 Why Use It?**

1. ✅ **No boilerplate code** – You don't need to write SQL or even basic queries.
2. ✅ **Built-in methods** – save(), findAll(), deleteById(), existsById(), etc.
3. ✅ **Auto-generated queries** – You can write methods like findByName, findByEmailContaining, etc., and Spring does the rest.

**🔍 Common Methods from JpaRepository:**

| **Method** | **Description** |
| --- | --- |
| save(entity) | Inserts or updates an entity |
| findById(id) | Finds by primary key |
| findAll() | Returns all rows |
| deleteById(id) | Deletes by primary key |
| count() | Returns number of rows |
| existsById(id) | Checks if entity exists |

**🧩 Related Interfaces:**

* CrudRepository<T, ID> – Basic CRUD operations.
* PagingAndSortingRepository<T, ID> – Adds pagination and sorting.
* JpaRepository<T, ID> – Includes all of the above, plus JPA-specific features.

The Service Layer

* The service layer acts as a bridge between the persistence layer (responsible for data access) and the presentation layer (handling user interaction)
* It encapsulates the business logic of the application, orchestrates interactions between different components, and provides a clean interface for external clients to interact with the system.
* By abstracting away, the complexities of data access and business operations, the service layer promotes modularity, maintainability, and scalability

Reflection

* **Reflection** in Java is a powerful feature that allows a program to **inspect and manipulate classes, methods, fields, and constructors at runtime**, even if their names or behavior were not known at compile time.
* It belongs to the java.lang.reflect package.

### ✅ ****Why Use Reflection?****

* To inspect class structure (fields, methods, annotations, etc.)
* To instantiate objects dynamically
* To call methods or access fields dynamically
* To build frameworks like Spring, Hibernate, JUnit, etc.

import java.lang.reflect.Method;

public class Demo {

public void sayHello() {

System.out.println("Hello!");

}

public static void main(String[] args) throws Exception {

// Get the Class object

Class<?> clazz = Class.forName("Demo");

// Create an object of Demo

Object obj = clazz.getDeclaredConstructor().newInstance();

// Get the method

Method method = clazz.getMethod("sayHello");

// Call the method

method.invoke(obj); // Output: Hello!

}

}

**🧠 Key Reflection Concepts**

| **Concept** | **Description** |
| --- | --- |
| Class<?> | Represents a class in Java |
| Field | Represents a class field (variable) |
| Method | Represents a method |
| Constructor | Represents a constructor |
| .invoke() | Call a method dynamically |
| .get() / .set() | Read/write field values dynamically |

### Disadvantages of Reflection

* **Slower**: Reflection bypasses some compiler optimizations.
* **Security Risks**: Can access private fields/methods if not handled carefully.
* **Harder to Maintain**: Dynamic code can be confusing and error-prone.
* **Breaks Encapsulation**: Can modify private fields or call private methods.

Annotation for Validations

* @NotNull
  + Ensures that the annotated field is not null.
* @NotEmpty
  + Ensures that the annotated field is not null and its size/length is greater than zero. (For collections, arrays, and Strings)
* @NotBlank
  + Ensures that the annotated string is not null and its trimmed length is greater than zero
* @size
  + Validates that the annotated element’s size falls within the specified range

Benefits for Exception Handling

* Prevent application crashes
* Provide user-friendly error responses
* Facilitate debugging and maintenance
* Ensure consistent error handling across the application

Transforming API Response

* Extend your class with @ResponseBodyAdvice<Object> to define the custom return type
* Use @RestControllerAdvice for global API Response transformation
* Return appropriate HTTP status codes and error messages
* You can also return the timestamp of the API response