Introduction to Web Development and REST APIs

Iyed ZAIRI

International Institute of Technology

Federative Project

What is Web Development?

- Web development is the process of building and maintaining websites.
- It involves both the front-end (what users see) and back-end (server-side logic) of web applications.
- Web development can be static or dynamic, with dynamic sites making use of databases and APIs.
- Key technologies for web development:
  - HTML, CSS, JavaScript (front-end)
  - Server-side programming languages like Java, Python, Ruby (back-end)
  - APIs for communication between systems

Introduction to RESTful Services

- REST (Representational State Transfer) is an architectural style for designing networked applications.
- It uses standard HTTP methods and is stateless, meaning each request contains all the information needed to process it.
- REST APIs are commonly used for communication between front-end and back-end services.
- Key principles of REST:
  - Stateless communication
  - Client-server architecture
  - Use of standard HTTP methods

HTTP Methods: GET, POST, PUT, DELETE

- **GET**: Retrieve data from a server (e.g., requesting a webpage or retrieving user details).
- **POST**: Send data to the server to create a resource (e.g., submitting a form or adding a new user).
- **PUT**: Update an existing resource on the server (e.g., updating user details).
- **DELETE**: Remove a resource from the server (e.g., deleting a user or post).

Understanding HTTP Status Codes

- HTTP status codes are three-digit numbers sent by the server to indicate the outcome of the request.
- **2xx**: Successful responses
  - 200 OK: The request was successful.
  - 201 Created: The resource was successfully created.
- **4xx**: Client errors
  - 400 Bad Request: The request was malformed or invalid.
  - 404 Not Found: The requested resource could not be found.
- **5xx**: Server errors
  - 500 Internal Server Error: An error occurred on the server while processing the request.

Basic REST API Concepts

- **Endpoints**: A URL where the client can interact with a service (e.g., `https://api.example.com/users`).
- **Requests**: The action made by the client to communicate with the server, including method type (GET, POST, etc.), headers, and body data.
- **Responses**: The server's reply to a client's request, including status code and data (often in JSON or XML format).

Installing Java and Spring Boot

- To work with Spring Boot, you need to have Java installed on your machine.
- Download and install the latest version of the Java Development Kit (JDK) from the official Oracle website or OpenJDK.
- To check if Java is installed, open a terminal or command prompt and run:
  - `java -version`
- Spring Boot requires at least JDK 8, but the latest versions of Spring Boot work with JDK 11 or higher.
- Download and install Spring Boot from `https://spring.io/projects/spring-boot`, or use Spring Initializr (discussed later) to generate projects.

Setting Up IntelliJ IDEA

- Download and install IntelliJ IDEA from `https://www.jetbrains.com/idea/download/`.
- Choose the **Ultimate** version for full Spring Boot support (or the free Community edition for basic features).
- After installation, open IntelliJ and configure your JDK:
  - Go to `File > Project Structure > Project`.
  - Set the `Project SDK` to the installed JDK version.
- Install the Spring Boot plugin in IntelliJ IDEA for enhanced features.
  - Go to `File > Settings > Plugins`.
  - Search for "Spring Boot" and install the plugin.

Introduction to Maven/Gradle for Dependency Management

- **Maven** and **Gradle** are tools used for managing dependencies and automating the build process in Java projects.
- Both tools allow you to specify dependencies (such as Spring Boot) in a configuration file.
- **Maven**: Uses `pom.xml` for dependency management.
  - Example Maven dependency for Spring Boot:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifac
</dependency>
```

- **Gradle**: Uses `build.gradle` for dependency management.
  - Example Gradle dependency for Spring Boot:

```
implementation 'org.springframework.boot:spring-
```

- Spring Initializr can generate projects with pre-configured Maven or Gradle files.

Creating Your First Spring Boot Project

- Use **Spring Initializr** (https://start.spring.io/) to generate your first Spring Boot project.
  - Select the project type: Maven or Gradle.
  - Choose Java version, Spring Boot version, and other options (dependencies like "Spring Web" or "Spring Data JPA").
  - Download the generated project as a ZIP file and unzip it.
- Open the project in IntelliJ IDEA.
  - Go to `File > Open` and select the unzipped folder.
  - IntelliJ will automatically detect the project and ask to import the Maven/Gradle project.
- Run your first Spring Boot application:
  - Locate the main application class (e.g., `YourApplication.java`).
  - Right-click on the class and choose `Run`.
  - The Spring Boot application will start on the default port (usually 8080).
- You can now access the application in your browser at `http://localhost:8080`.

Building Your First REST API with Spring Boot

- **Understanding the Anatomy of a Spring Boot Application**
  - A Spring Boot application consists of:
    - A main class annotated with `@SpringBootApplication`
    - A controller class to handle HTTP requests
    - Configuration classes (optional)

Creating a Simple Controller with `@RestController`

- Use `@RestController` to define REST API endpoints
- Automatically serializes responses to JSON format
- Example:

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

Mapping HTTP Requests with `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc.

- `@RequestMapping` – A general-purpose annotation for mapping HTTP requests
- `@GetMapping` – Specifically for handling GET requests
- `@PostMapping` – For POST requests
- `@PutMapping` – For PUT requests
- Example:

```
@GetMapping("/greeting")
public String greet(@RequestParam String name) {
    return "Hello, " + name + "!";
}
```

Handling Request Parameters and Body

- Handle request parameters using `@RequestParam` or `@PathVariable`
- Handle request body with `@RequestBody`
- Example for `@RequestBody`:

```
@PostMapping("/subject")
public User createSubject(@RequestBody Subject subj
    return subjectService.save(subject);
}
```

Returning Responses: `ResponseEntity` and JSON

- Return responses with `ResponseEntity` for better control over HTTP status codes
- Automatically serialize Java objects into JSON format
- Example:

```
@GetMapping("/subject/{id}")
public ResponseEntity<Subject> getSubject(@PathVari
    Subject subject = subjectService.findById(id);
    if (subject == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(subject);
}
```

Connecting to a Database (JPA and MySQL)

- Introduction to JPA (Java Persistence API)
- Creating Entity Classes
- Setting Up MySQL Database
- Performing CRUD Operations with Native SQL
- Transitioning to JPA ORM

Introduction to JPA (Java Persistence API)

- JPA is a specification for ORM (Object-Relational Mapping).
- It abstracts SQL queries and makes database interactions easier.
- Spring Boot uses **Spring Data JPA** to implement JPA.
- Before using JPA, we will first work with raw SQL.

Setting Up MySQL Database

- Install and configure MySQL server.
- Create a new database:

```
CREATE DATABASE mydb;
```

- Configure `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.database-platform=org.hibernate.dialect.
MySQL8Dialect
```

Performing CRUD Operations with Native SQL

- Define a repository using `@Repository` and `JdbcTemplate`.
- Example: Insert User into MySQL

```
@Repository
public class UserRepository {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void saveUser(String name, String email) {
        String sql = "INSERT INTO users (name, email)
        VALUES (?, ?)";
        jdbcTemplate.update(sql, name, email);
    }
}
```

Fetching Data with Native SQL

```
public List<User> getAllUsers() {
    String sql = "SELECT * FROM users";
    return jdbcTemplate.query(sql,
    new BeanPropertyRowMapper<>(User.class));
}
```

- Here, BeanPropertyRowMapper maps SQL result to User object.

Transitioning to JPA ORM

- Replace raw SQL with JPA entities.
- Example: Defining an entity class.

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Getters and Setters
}
```

Using Spring Data JPA for CRUD

- Define a repository interface.

```
@Repository
public interface UserRepository extends JpaRepository<User,
Long> {
    List<User> findByName(String name);
}
```

- Spring Data JPA generates SQL queries automatically.

Types of Entity Relationships

- **One-to-Many** (e.g., User $\rightarrow$ Posts)
- **Many-to-One** (e.g., Post $\rightarrow$ User)
- **Many-to-Many** (e.g., Student Course)
- Relationships are defined using annotations:
  - `@OneToMany`, `@ManyToOne`
  - `@ManyToMany`, `@JoinColumn`, `@JoinTable`

One-to-Many Example: User → Posts

```
@Entity
public class User {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL
    private List<Post> posts;
}

@Entity
public class Post {
    @Id @GeneratedValue
    private Long id;
    private String content;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
}
```

Many-to-One: Post → User

- Many posts can belong to one user.
- `@ManyToOne` on the child side
- `@OneToMany` on the parent with `mappedBy`

Important: The `mappedBy` attribute tells JPA which side owns the relationship.

Many-to-Many: Student Course

```
@Entity
public class Student {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id"
    )
    private List<Course> courses;
}
```

```java
@Entity
public class Course {
    @Id @GeneratedValue
    private Long id;
    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
}
```

Cascading and Fetching Strategies

- `CascadeType`: Specifies operations to cascade (e.g., `ALL`, `PERSIST`, `REMOVE`)
- `FetchType.LAZY`: Load relationship on demand (default for collections)
- `FetchType.EAGER`: Load relationship immediately (default for `@ManyToOne`)

Example:

```
@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZ
private List<Post> posts;
```