**Spring Boot - Introduction to RESTful Web Services**

**RESTful Web Services REST** stands for **REpresentational State Transfer**. It was developed by **Roy Thomas Fielding**. REST was an **architectural approach** designed to make the optimum use of the **HTTP protocol**. A resource can be anything, it can be accessed through a **URI (Uniform Resource Identifier)**. Unlike SOAP, REST does not have a standard messaging format. We can build REST web services using many representations. REST is a style whose purpose is to constrain our architecture to a client-server architecture and is designed to use stateless communication protocols like HTTP.

The main methods of HTTP we build web services for are:

1. **GET**: Reads existing data.

The default request method for HTTP. We don't have any request body with this method, but we can define multiple request parameters or path variables in the URL. This method is used for obtaining some resources.

*@GetMapping("/user/{userId}")  
public ResponseEntity<Object> getUser(@PathVariable int userId) {  
UserEntity user = userService.getUser(userId);  
return new ResponseEntity<>(user, HttpStatus.OK);  
}*

1. **POST**: Creates new data

The POST method of HTTP is used to create a resource. We have a request body in this method and can also define multiple request parameters or path variables in the URL.

*@PostMapping(value = "/user")  
public ResponseEntity<Object> addUser(@RequestBody UserEntity user) {  
userService.saveOrUpdate(user);  
return new ResponseEntity<>("User is created successfully", HttpStatus.CREATED);  
}*

1. **PUT**: Updates existing data.

The PUT method of HTTP is used to update an existing resource. We have a request body in this method and can also define multiple request parameters or path variables in the URL.

*@PutMapping("/user/{userId}")  
public ResponseEntity<Object> updateUser(@PathVariable int userId, @RequestBody UserEntity user) {  
userService.saveOrUpdate(user);  
return new ResponseEntity<>("User is updated successfully", HttpStatus.OK);  
}*

1. **DELETE**: Deletes the data.

The DELETE method of HTTP is used to remove a resource. We don't have a request body in this method but can define multiple request parameters or path variables in the URL.

*@DeleteMapping("/user/{userId}")  
public ResponseEntity<Object> deleteUser(@PathVariable int userId) {  
userService.deleteUser(userId);  
return new ResponseEntity<>("User is deleted successfully", HttpStatus.OK);  
}*

## HTTP Standard Status Codes

The status codes defined in HTTP are the following:

* **200**: Success
* **201**: Created
* **401**: Unauthorized
* **404**: Resource Not Found
* **500**: Server Error

The web services are completely stateless. The service producer and consumer have a mutual understanding of the context and content being passed along. Idempotent requests like GET requests are considered cacheable, modern caching mechanisms allow POST requests to be cached under certain conditions when using **Cache-Control headers**.

### REST API Security

Modern RESTful APIs should follow the below security practices:

* **Authentication and Authorization:** Use OAuth 2.0 or JWT-based authentication.
* **Rate Limiting:** Implement API rate limiting to prevent abuse.
* **Input Validation:** Sanitize input to prevent SQL injection and XSS attacks.
* **HTTPS Enforcement**: Ensure secure communication by enforcing HTTPS.

**Records**

Java records  were intended to be used as a quick way to create data carrier classes, i.e. the classes whose objective is to simply contain data and carry it between modules, also known as POJOs (Plain Old Java Objects) and DTOs (Data Transfer Objects).

*public record Person (String name, String address) {}*

**Text Blocks**

***String*** *textBlock = """ Hello, this is a multi-line text block. """;*

**Switch Expressions**

Java 12 introduced switch expressions which evaluate a single value, and can be used in statements. Instead of combining nested if–else-operators (?:), we can now use a switch–case-construct:

*DayOfWeek day = DayOfWeek.FRIDAY;*

*int numOfLetters = switch (day) {*

*case MONDAY, FRIDAY, SUNDAY -> 6;*

*case TUESDAY -> 7;*

*case THURSDAY, SATURDAY -> 8;*

*case WEDNESDAY -> 9;*

*};*

**Sealed Classes and Interfaces**

Sealed classes can limit inheritance by specifying allowed subclasses:

***public******abstract******sealed******class******Pet******permits*** *Dog, Cat {}*

Spring Boot is a powerful framework that makes it easy to create RESTful APIs. Spring Boot simplifies REST API development by providing built-in features such as automatic configurations, embedded servers, and minimal boilerplate code.

## Step-by-Step Implementation to Create REST API using Spring Boot

Below are the steps to implement REST API in Spring Boot with MySQL and JPA.

### Step 1: Setup Development Environment

### Step 2: Create a Spring Boot Project

The first step is to create a new Spring Boot project using the Spring Initializr.

Click on the “Add Dependencies” button and add the following dependencies:

* Spring Web
* Spring Data JPA
* MySQL Driver

### Step 3: Add Dependencies

To create a REST API with Spring Boot, include the following dependencies in pom.xml.

### Step 4: Configure Database in application.properties

For MySQL configuration, add the following to src/main/resources/application.properties:

*spring.datasource.url=jdbc:mysql://localhost:3306/mydb*

*spring.datasource.username=root*

*spring.datasource.password=yourpassword*

*spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver*

*spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect*

*spring.jpa.hibernate.ddl-auto=update*

### Step 5: Create Entity Class

we need to create an entity class to represent our data model. In this example, we will create a “Product” entity class.

**package** **com.example.entity**;

**import** **javax.persistence.\***;

@Entity

@Table(name = "product")

**public** **class** **Product** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** Long id;

@Column(nullable = **false**)

**private** String name;

@Column(nullable = **false**)

**private** double price;

@Column(nullable = **false**)

**private** int quantity;

**public** Long getId() {

**return** id;

}

**public** String getName() {

**return** name;

}

**public** double getPrice() {

**return** price;

}

**public** int getQuantity() {

**return** quantity;

}

**public** void setId(Long id) {

**this**.id = id;

}

**public** void setName(String name) {

**this**.name = name;

}

**public** void setPrice(double price) {

**this**.price = price;

}

**public** void setQuantity(int quantity) {

**this**.quantity = quantity;

}

}

* @Entity-- specifies that this class is an entity and should be mapped to a database table.
* @Table-- specifies the name of the database table that will store instances of this entity.
* @Id-- specifies the primary key of the entity.
* @GeneratedValue-- specifies that the primary key should be generated automatically.

### Step 6: Create Repository Interface

Now, we need to create a repository interface to handle database operations for the “Product” entity.

**package** **com.example.springbootapi.repository**;

**import** **com.example.springbootapi.entity.Product**;

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

**import** **org.springframework.stereotype.Repository**;

@Repository

**public** **interface** **ProductRepository** **extends** JpaRepository<Product, Long> {

}

* @Repository--specifies that this interface is a repository, and Spring will create an instance of it automatically.