4/25/2024

Creating API and Microservices with Java Spring Boot

Contents

Spring 6 REST services

Setup notes:
1. Spring 6 and JDK compatibility https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-Versions
2. Install jdk 17 and eclipse
3. Maven dependencies

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>3.2.5</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>demo1</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>demo1</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>17</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>

                <dependency>
                  <groupId>org.springframework</groupId>
                  <artifactId>spring-webmvc</artifactId>
                </dependency>

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

```
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>

</project>
```

- Parent POM: The spring-boot-starter-parent POM provides default configuration for Maven, reducing the need to specify version numbers for various dependencies.
- Dependencies: The spring-boot-starter-web dependency includes all necessary components for building web and RESTful applications, including Spring MVC, and default Tomcat as the embedded server.
- Java Version: The property java.version is set to 17 to ensure compatibility with Java 17.
- Build Plugin: The spring-boot-maven-plugin helps in packaging and running the Spring Boot application.

Note: Using eclipse, you can install Spring using the eclipse marketplace. This allows you to directly create spring boot projects.
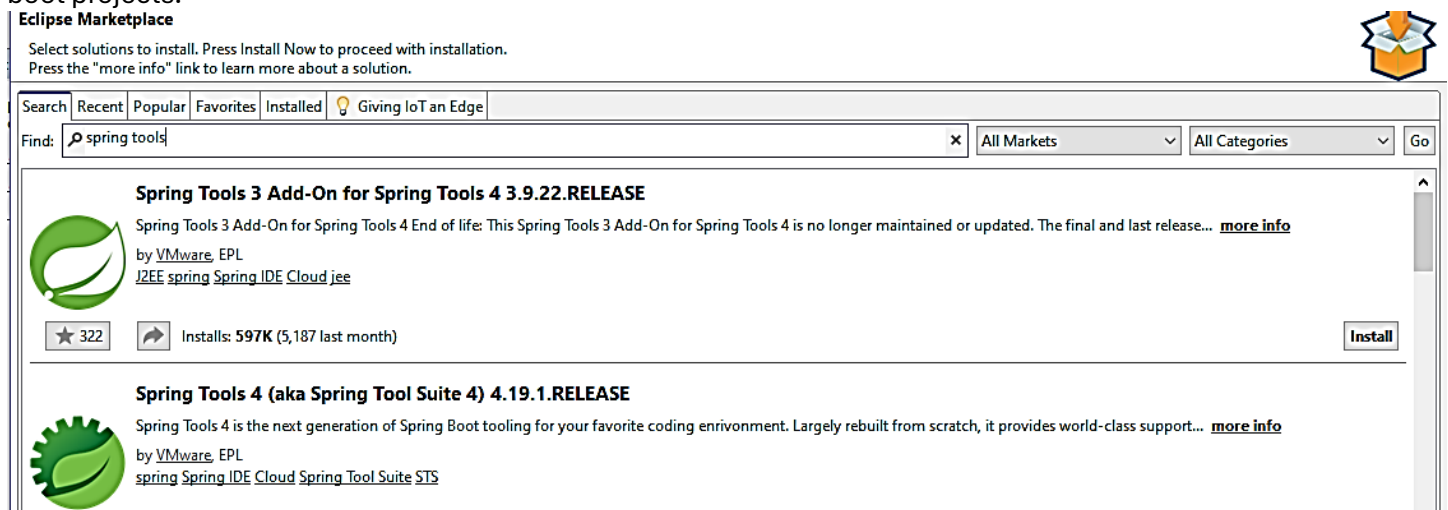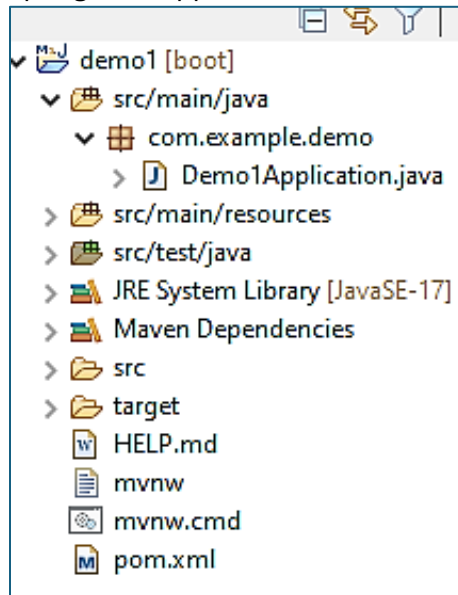

Get more information here:

https://start.spring.io/

Create Spring Boot project from Eclipse

File→New→Spring→New Spring Starter Project

Spring Boot Application

```
v demo1 [boot]
  v src/main/java
    v com.example.demo
      > Demo1Application.java
  > src/main/resources
  > src/test/java
  > JRE System Library [JavaSE-17]
  > Maven Dependencies
  > src
  > target
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml
```

```java
package com.example.demo;

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

@SpringBootApplication
public class Demo1Application {

        public static void main(String[] args) {
                SpringApplication.run(Demo1Application.class, args);
        }

}
```
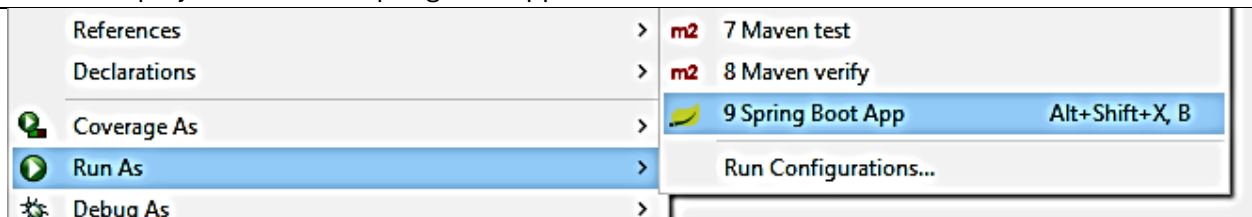
The @SpringBootApplication annotation is a convenience annotation that adds:
- @Configuration: Tags the class as a source of bean definitions for the application context.
- @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- @ComponentScan: Tells Spring to look for other components, configurations, and services in the com.example.demo package, allowing it to find the controllers.
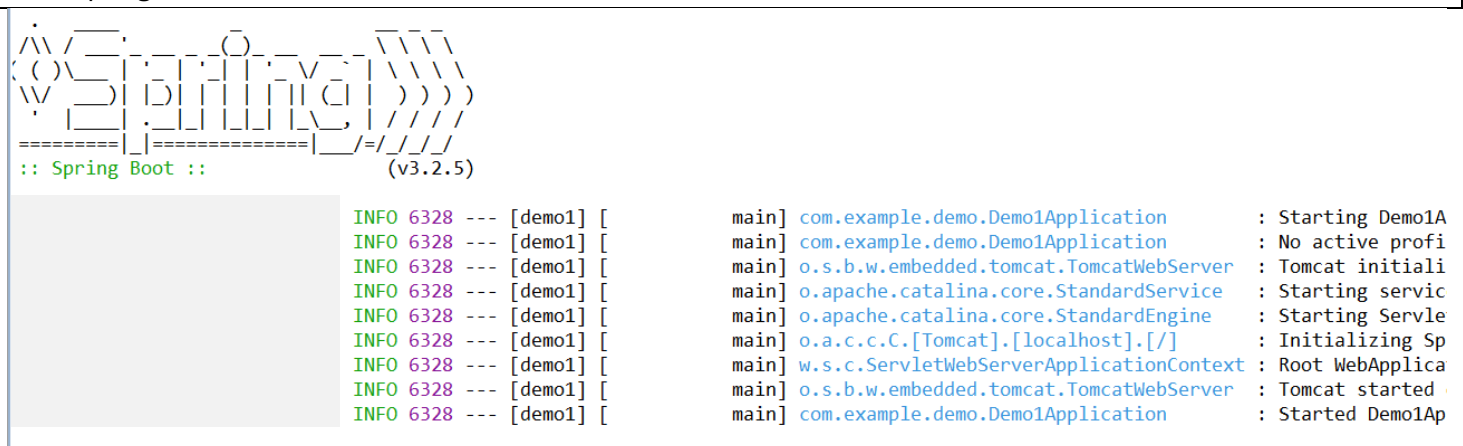
Run the project as a Spring Boot Application:

Rclick the project→run as→spring boot app

```
References              >   m2  7 Maven test
Declarations            >   m2  8 Maven verify
Coverage As             >       9 Spring Boot App        Alt+Shift+X, B
Run As                  >       Run Configurations...
Debug As                >
```

You can also run the project in the terminal

```
mvn spring-boot:run
```

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.2.5)

INFO 6328 --- [demo1] [         main] com.example.demo.Demo1Application       : Starting Demo1A
INFO 6328 --- [demo1] [         main] com.example.demo.Demo1Application       : No active profi
INFO 6328 --- [demo1] [         main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initiali
INFO 6328 --- [demo1] [         main] o.apache.catalina.core.StandardService  : Starting servic
INFO 6328 --- [demo1] [         main] o.apache.catalina.core.StandardEngine   : Starting Servle
INFO 6328 --- [demo1] [         main] o.a.c.c.C.[Tomcat].[localhost].[/]      : Initializing Sp
INFO 6328 --- [demo1] [         main] w.s.c.ServletWebServerApplicationContext : Root WebApplica
INFO 6328 --- [demo1] [         main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started
INFO 6328 --- [demo1] [         main] com.example.demo.Demo1Application       : Started Demo1Ap
```

Note: Application Properties File
You can specify the server port in the application.properties or application.yml file located in the src/main/resources directory. This is the most common way to set configuration properties in Spring Boot.

Using application.properties:

```
server.port=8081
```

Using the @Controller and @RestController

In Spring 6.0, which is part of the Spring Framework, the @Controller and @RestController annotations are used to define controllers, but they serve slightly different purposes and are used in different types of applications.

@Controller
- Purpose: The @Controller annotation is used in Spring MVC to build web applications. It indicates that a particular class serves the role of a controller in the MVC (Model-View-Controller) pattern.
- Behavior: A class annotated with @Controller is capable of handling requests and returning a response typically in the form of a view (like a JSP or a template engine-rendered page), though it can also return other types of responses.
- Response Handling: Methods in a @Controller-annotated class often return a String indicating a view name, which the Spring view resolver will use to render the HTML content. If you want to return a response body in a RESTful manner from a @Controller, you need to annotate the method with @ResponseBody to indicate that the return type should be written directly to the HTTP response body.

@RestController
- Purpose: Introduced as part of the Spring 4 release, the @RestController annotation simplifies the creation of RESTful web services. It is a convenience annotation that combines @Controller and @ResponseBody.
- Behavior: A class annotated with @RestController is also a controller, but it is specifically intended for RESTful web services. It implies that every method inherits the @ResponseBody annotation and therefore the method return type will automatically be written directly to the HTTP response body.
- Response Handling: There's no need to use @ResponseBody on any method within a @RestController because it's assumed by default. Methods typically return data objects (like POJOs or collections), which Spring automatically converts to JSON or XML.

Key Differences
- View vs. Data: @Controller is typically used where you want to develop a web application that renders server-side generated HTML. @RestController is used when you want to develop a service that returns data directly (e.g., JSON or XML) for its clients, like modern web applications using Angular or React, or when building microservices.
- Annotation Requirements: In @Controller, you often need to annotate response-producing methods with @ResponseBody (or use @ResponseEntity) to send JSON or XML directly to the client, whereas in @RestController, all methods assume @ResponseBody semantics.

Demo: Add a new controller class.
Note: You might not see any results yet so we will construct a Controller class. **All controller class names should be appended with the word Controller**. Example: "HelloWorldController"

HelloWorldController.java

```java
package com.example.demo;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

//@Controller
//@ResponseBody
//or simply
@RestController
public class HelloWorldController {

        // defined http get method can also work with a defined url
        // http://localhost:8080/hello-world
        @GetMapping("/hello-world")
        public String helloWorld() {
                return "Hello World";
        }

}
```

Test it on the browser http://localhost:8080/hello-world

<mark>Demo: Build a RESTful API that returns JSON</mark>
1. Create a class that represents the object you need to return as a response

Employee.java

```java
package com.example.demo;

public class Employee {

        private String firstName;
        private String lastName;

        //rclick->source->generate constructor from fields
        public Employee(String firstName, String lastName) {
                super();
                this.firstName = firstName;
                this.lastName = lastName;
        }


        //rclick->source->generate getters and setters
        public String getFirstName() {
                return firstName;
        }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
```

```
        }
        public String getLastName() {
                return lastName;
        }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }

}
```

2. Create a controller class

EmployeeController.java

```
package com.example.demo;

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

@RestController
public class EmployeeController {

        //http://localhost:8080/employee
        @GetMapping("/employee")
        public Employee getEmployee() {
                return new Employee("john", "doe");
        }
}
```

3. Run the REST API and try accessing the endpoint http://localhost:8080/employee

Demo: Build a RESTful API that returns a list

1. From the previous activity, modify the EmployeeController.java

```
package com.example.demo;

import java.util.ArrayList;
import java.util.List;

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

@RestController
public class EmployeeController {

        //http://localhost:8080/employee
        @GetMapping("/employee")
        public Employee getEmployee() {
                return new Employee("john", "doe");
        }
```

```
        //http://localhost:8080/employees
        @GetMapping("/employees")
        public List<Employee> getEmployees(){
                List<Employee> employees = new ArrayList<>();
                employees.add(new Employee("Kevin", "Sanchez"));
                employees.add(new Employee("Mike", "Morey"));
                employees.add(new Employee("Lisa", "Downey"));
                return employees;
        }
}
```

Structures of REST implementation

Implementing GET, POST, PATCH, PUT, and DELETE RESTful services

1. Create the Book Model

In the src/main/java/com/example/bookstore directory, create a model class Book.java:

```java
package com.example.bookstore.model;

public class Book {
  private Integer id;
  private String title;
  private String author;

  public Book() {
  }

  public Book(Integer id, String title, String author) {
    this.id = id;
    this.title = title;
    this.author = author;
  }

  public Integer getId() {
    return id;
  }

  public void setId(Integer id) {
    this.id = id;
  }

  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }
```

```
  public String getAuthor() {
    return author;
  }

  public void setAuthor(String author) {
    this.author = author;
  }
}
```

2. Create the BookController

Create a REST controller in the src/main/java/com/example/bookstore/controller directory:

```
package com.example.bookstore.controller;

import com.example.bookstore.model.Book;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

@RestController
@RequestMapping("/api/books")
public class BookController {
  private final List<Book> books = new ArrayList<>();
  private final AtomicInteger counter = new AtomicInteger();

  @GetMapping
  public List<Book> getAllBooks() {
    return books;
  }

  @PostMapping
  public Book addBook(@RequestBody Book book) {
    book.setId(counter.incrementAndGet());
    books.add(book);
    return book;
  }

  @GetMapping("/{id}")
  public Book getBookById(@PathVariable int id) {
    return books.stream()
        .filter(book -> book.getId().equals(id))
        .findFirst()
        .orElse(null);
  }

  @PutMapping("/{id}")
  public Book updateBook(@PathVariable int id, @RequestBody Book updatedBook) {
    Book book = books.stream()
```

```
        .filter(b -> b.getId() == id)
        .findFirst()
        .orElse(null);
    if (book != null) {
        book.setTitle(updatedBook.getTitle());
        book.setAuthor(updatedBook.getAuthor());
    }
    return book;
}

@PatchMapping("/{id}")
public Book patchBook(@PathVariable int id, @RequestBody Book updatedBook) {
    Book book = books.stream()
        .filter(b -> b.getId() == id)
        .findFirst()
        .orElse(null);
    if (book != null) {
        if (updatedBook.getTitle() != null) book.setTitle(updatedBook.getTitle());
        if (updatedBook.getAuthor() != null) book.setAuthor(updatedBook.getAuthor());
    }
    return book;
}

@DeleteMapping("/{id}")
public void deleteBook(@PathVariable int id) {
    books.removeIf(b -> b.getId() == id);
}
}
```
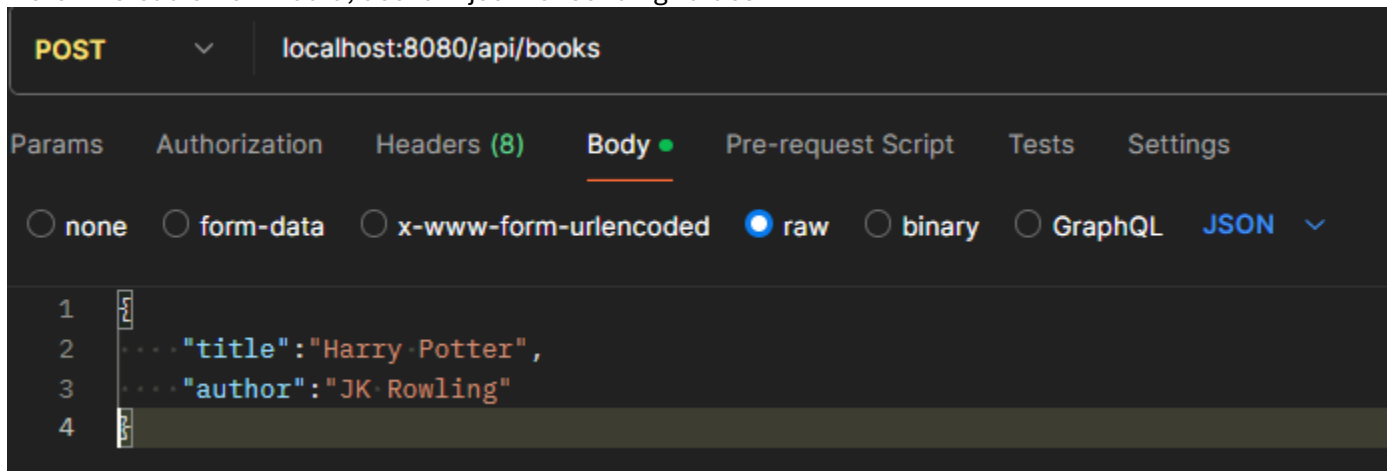
Note: PUT vs PATCH
- Bandwidth: If only a small part of the resource needs to change, PATCH requests typically require less bandwidth because they only transmit the changes, not the complete resource.
- Use Case: Use PUT when you want to replace a resource entirely and have all of its representations sent to the server. Use PATCH when you want to make specific changes to parts of the resource without affecting the whole.

Note: instead of form-data, use raw-json for sending values

POST ∨ localhost:8080/api/books

| Params | Authorization | Headers (8) | **Body •** | Pre-request Script | Tests | Settings |

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON ∨

```
1  {
2      "title":"Harry Potter",
3      "author":"JK Rowling"
4  }
```

Use of request parameters

1. Modify the EmployeeController.java to add a new method

```java
package com.example.demo;

...

@RestController
public class EmployeeController {

        ...

        //Rest API Endpoint to handle query parameter
        //http://localhost:8080/employee/query?firstName=Kevin&lastName=Turney
        @GetMapping("/employee/query")
        public Employee employeeRequestParameter(
                    @RequestParam(name = "firstName") String firstName,
                    @RequestParam(name = "lastName") String lastName
                    )
        {
                return new Employee(firstName,lastName);
        }
}
```

Use of path variables

Note: Path Variables aka Path Parameters are values that are submitted from a URL

1. Modify the EmployeeController.java and add another method

```java
package com.example.demo;
...

@RestController
public class EmployeeController {

        ...

        //http://localhost:8080/employee/steph/curri
        //to bind URI template variable to method variable, use @PathVariable
        @GetMapping("/employee/{firstName}/{lastName}")
        public Employee employeePathVariable(
                    @PathVariable("firstName") String firstName,
                    @PathVariable("lastName")String lastName
                    )
        {
                return new Employee(firstName,lastName);

        }
}
```

Dealing with Media Types

In web development, especially in REST APIs, handling different media types such as XML and JSON is crucial for allowing clients to interact with your service in a variety of formats. Spring Boot makes it easy to produce and consume these media types.

    1. Setup Dependencies

To support JSON, Spring Boot uses Jackson by default, which is included in the spring-boot-starter-web dependency. For XML support, you need to add an additional dependency for JAXB (Java Architecture for XML Binding).

For Maven, add this to your pom.xml:

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

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

            <dependency>
                    <groupId>com.fasterxml.jackson.dataformat</groupId>
                    <artifactId>jackson-dataformat-xml</artifactId>
            </dependency>
...
```

    2. Create Model Classes

Create a simple model class that can be serialized to JSON and XML. Here's an example of a simple Book class.

```java
package com.example.demo.model;
import jakarta.xml.bind.annotation.XmlRootElement;

@XmlRootElement  // Makes this class ready for JAXB XML serialization
public class Book {
  private String title;
  private String author;

  // Default constructor is necessary for XML deserialization
  public Book() {
  }

  public Book(String title, String author) {
    this.title = title;
    this.author = author;
  }
```

```java
  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  public String getAuthor() {
    return author;
  }

  public void setAuthor(String author) {
    this.author = author;
  }
}
```

3. Create a Controller

Now, create a controller that will handle requests and respond with either JSON or XML based on the Accept header in the request.

```java
package com.example.demo.controller;

import com.example.demo.model.Book;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Arrays;
import java.util.List;

@RestController
@RequestMapping("/api/books")
public class BookController {

  @GetMapping(produces = {MediaType.APPLICATION_XML_VALUE,MediaType.APPLICATION_JSON_VALUE})
  public List<Book> getAllBooks() {
    //return books;
        return Arrays.asList(
                    new Book("1984", "George Orwell"),
                    new Book("Brave New World", "Aldous Huxley")
                    );
  }
}
```

4. Configure Application Properties

Make sure your application is configured to support XML. This usually works out-of-the-box, but if there's any issue, you can explicitly set the property in application.properties:

properties

```
spring.mvc.contentnegotiation.favor-parameter=true
spring.mvc.contentnegotiation.media-types.xml=application/xml
```

5. Run the Application
6. Testing the API. Use a tool like Postman or a simple curl command to test your API.
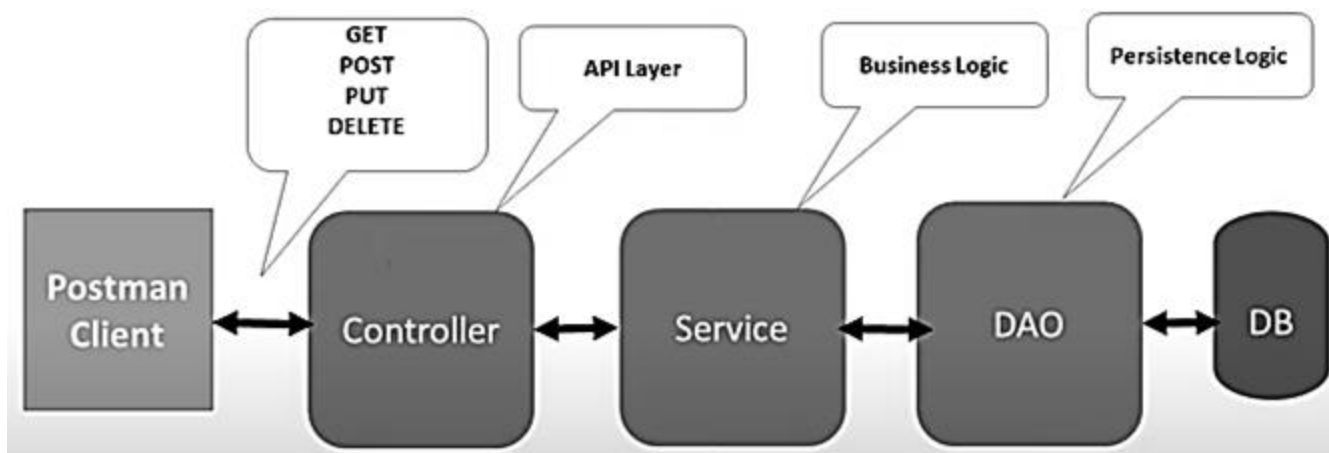
To request JSON:

```
curl -X GET http://localhost:8080/api/books -H "Accept: application/json"
```

To request XML:

```
curl -X GET http://localhost:8080/api/books -H "Accept: application/xml"
```

Spring Boot + Hibernate + MySQL CRUD REST API (Integration with Data Layer)

1. Create a database in MySQL
2. Create Spring Boot Project
    File→New→Spring→New Spring Starter Project

Configuration:

Project	Maven
Language	Java
Spring Boot	3.2.0
Group	com.restproject
Artifact	Rest1
Name	Rest1
Description	
Package Name	com.restproject.Rest1
Packaging	Jar
Java Version	20
Dependencies	Spring Web
	Spring Data JPA
	MySQL Driver
	Lombok

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>3.2.5</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>restcrud</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>restcrud</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>17</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jpa</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>

                <dependency>
                        <groupId>com.mysql</groupId>
                        <artifactId>mysql-connector-j</artifactId>
                        <scope>runtime</scope>
                </dependency>
                <dependency>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <optional>true</optional>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
```

```
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                        <configuration>
                                <excludes>
                                        <exclude>
                                                <groupId>org.projectlombok</groupId>
                                                <artifactId>lombok</artifactId>
                                        </exclude>
                                </excludes>
                        </configuration>
                </plugin>
        </plugins>
    </build>

</project>
```
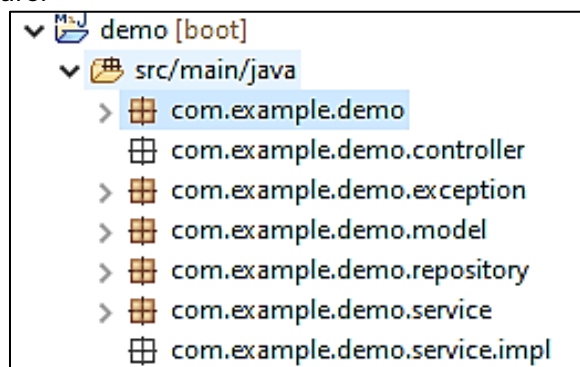
3. Create a packaging structure.

4. Modify the application.properties file

```
# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/ems
spring.datasource.username=john
spring.datasource.password=123
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
# spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver        #or you can just leave this and the preceding
                                                                     line active

# JPA Configuration
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update

# spring.autoconfigure.exclude = org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

5. If, problems occurs, try commenting out this dependency in pom.xml (uncomment later after first
    successful build)

```
...
                <!--<dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
            </dependency>-->
...
```

6. Create a JPA entity

Note:
- ✓ We will be using Lombok library (@Data) which will reduce boilerplate code like getters/setters, constructors, toString, and other required methods for java classes.
- ✓ We will also be using @Entity from the javax persistence package (if spring boot 2.7 or older) or Jakarta persistence package (if spring boot 3) to make this class a jpa entity
- ✓ Add contructor for the fields (except the autogenerated id)
- ✓ Add getter and setters (except for the autogenerated id)

Rclick the model package→new class ("Employee")

```java
package com.example.demo.model;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;

@Data
@Entity
@Table(name="employees")
public class Employee {

        //define primary key
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private long id;

        //set column definition
        @Column(name="first_name")
        private String firstName;

        @Column(name="last_name")
        private String lastName;

        @Column(name="email")
        private String email;

        // default (empty) constructor
        // rclick Source->generate constructor using fields (uncheck all fields)
```

```java
public Employee() {
        super();
}

// constructor that uses params
// rclick Source->generate constructor using fields (check all fields)
public Employee(String firstName, String lastName, String email) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
}

// create getters and setters for all our properties
// rclick Source-->generate getters and setters (check all fields
public long getId() {
        return id;
}

public void setId(long id) {
        this.id = id;
}

public String getFirstName() {
        return firstName;
}

public void setFirstName(String firstName) {
        this.firstName = firstName;
}

public String getLastName() {
        return lastName;
}

public void setLastName(String lastName) {
        this.lastName = lastName;
}

public String getEmail() {
        return email;
}

public void setEmail(String email) {
        this.email = email;
}
```

```
}
```

7. Create a NotFound Error Exception

Rclick Exception Package→new class ("ResourceNotFoundException")

ResourceNotFoundException.java

```java
package com.example.demo.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value=HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException{

	private static final long serialVersionUID = 1L;
	private String resourceName;
	private String fieldName;
	private Object fieldValue;

	//generate constructors for the fields
	public ResourceNotFoundException(String resourceName, String fieldName, Object fieldValue) {
		super(String.format("%s not found with %s : %s",resourceName,fieldName,fieldValue));
		this.resourceName = resourceName;
		this.fieldName = fieldName;
		this.fieldValue = fieldValue;
	}

	//generate getters for the properties
	public String getResourceName() {
		return resourceName;
	}

	public String getFieldName() {
		return fieldName;
	}

	public Object getFieldValue() {
		return fieldValue;
	}
}
```

8. Create EmployeeRepository Interface

Rclick Repository Package→new Interface ("EmployeeRepository")

```
package com.example.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.model.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}
```

9. Create EmployeeService Interface

Rclick Service Package→new Interface ("EmployeeService")

```
package com.example.demo.service;

import com.example.demo.model.Employee;

public interface EmployeeService {
        Employee saveEmployee(Employee employee);
}
```

10. Create a class that implements the EmployeeService Interface

Rclick Service.impl Package→new Class ("EmployeeServiceImpl")

```
package com.example.demo.service.impl;

import org.springframework.stereotype.Service;

import com.example.demo.model.Employee;
import com.example.demo.repository.EmployeeRepository;
import com.example.demo.service.EmployeeService;

@Service
public class EmployeeServiceImpl implements EmployeeService {

        // dependency injection from the repository class
        private EmployeeRepository employeeRepository;

        // create constructor for this class
        public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
                super();
                this.employeeRepository = employeeRepository;
        }

        @Override
        public Employee saveEmployee(Employee employee) {
```

```
            return employeeRepository.save(employee);
        }
}
```

Note: You can easily implement unimplemented methods from the interface that is being extended by hovering the mouse over the class name then clicking "Add unimplemented methods"

```
1 package com.example.demo.service.impl;
2
3⊝import org.springframework.stereotype.Service;
4 import com.example.demo.service.EmployeeService;
5
6 @Service
7 public class EmployeeServiceImpl implements EmployeeService {
8        The type EmployeeServiceImpl must implement the inherited abstract method
9            EmployeeService.saveEmployee(Employee)
0        2 quick fixes available:
1          ⤸ Add unimplemented methods
```

Adding new records

11. Create a controller class that will provide mapping of API endpoints
Rclick Controller package→new class ("EmployeeController")
EmployeeController.java

```
package com.example.demo.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.model.Employee;
import com.example.demo.service.EmployeeService;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

        private EmployeeService employeeService;

        public EmployeeController(EmployeeService employeeService) {
                super();
                this.employeeService = employeeService;
        }

        // create API Endpoints
```

```
        // save new record
        @PostMapping()
        public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee){
                return new ResponseEntity<Employee>(
                                employeeService.saveEmployee(employee),
                                HttpStatus.CREATED
                                );
        }
}
```

12. Run the project and try sending JSON Post using PostMan

| POST | ⌄ | http://localhost:8080/api/employees |

Params   Authorization   Headers (8)   Body ●   Pre-request Script   Tests   Settings

○ none   ○ form-data   ○ x-www-form-urlencoded   ● raw   ○ binary   ○ GraphQL   JSON ⌄

```
1  {
2  ····"firstName":·"John",
3  ····"lastName":·"Goh",
4  ····"email":·"jg@abc.com"
5  }
```

Body   Cookies   Headers (5)   Test Results   ⊕ Status: 201 Created Ti

Pretty   Raw   Preview   Visualize   JSON ⌄   ⇆

```
1  {
2      "id": 4,
3      "firstName": "John",
4      "lastName": "Goh",
5      "email": "jg@abc.com"
```

Getting all records

13. Modify the EmployeeService.java class to add new service method
EmployeeService.java

```
package com.example.demo.service;

import java.util.List;

import com.example.demo.model.Employee;

public interface EmployeeService {
```

```
        Employee saveEmployee(Employee employee);
        List<Employee> getAllEmployees();
}
```

14. Modify the EmployeeServiceImpl.java to add the unimplemented method

```
 1 package com.example.demo.service.impl;
 2
 3⊕import org.springframework.stereotype.Service;⬚
 8
 9 @Service
10 public class EmployeeServiceImpl implements EmployeeService {
11           The type EmployeeServiceImpl must implement the inherited abstract method
             EmployeeService.getAllEmployees()
12      // depend  2 quick fixes available:
13      private E    ⮕ Add unimplemented methods
14
```

```
package com.example.demo.service.impl;

import java.util.List;

import org.springframework.stereotype.Service;

import com.example.demo.model.Employee;
import com.example.demo.repository.EmployeeRepository;
import com.example.demo.service.EmployeeService;


@Service
public class EmployeeServiceImpl implements EmployeeService {

        // dependency injection from the repository class
        private EmployeeRepository employeeRepository;

        // create constructor for this class
        public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
                super();
                this.employeeRepository = employeeRepository;
        }

        @Override
        public Employee saveEmployee(Employee employee) {
                return employeeRepository.save(employee);
        }

        @Override
        public List<Employee> getAllEmployees() {
```

```java
            // the code inside this method had been manually modified
            return employeeRepository.findAll();
        }

}
```

15. Modify the EmployeeController.java to add a RESTFul API Endpoint
EmployeeController.java

```java
package com.example.demo.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.model.Employee;
import com.example.demo.service.EmployeeService;
import com.example.demo.service.impl.EmployeeServiceImpl;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    private EmployeeService employeeService;

    public EmployeeController(EmployeeService employeeService) {
        super();
        this.employeeService = employeeService;
    }

    @GetMapping("/test")
    public String test() {
        return "controller ok";
    }

    // API Endpoints for CRUD operations
    // save new record
    @PostMapping("/new")
    public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee){
        return new ResponseEntity<Employee>(
```

```
                        employeeService.saveEmployee(employee),
                        HttpStatus.CREATED
                        );
        }

        // get all employees
        @GetMapping
        public List<Employee> getAllEmployee(){
                return employeeService.getAllEmployees();
        }
}
```

16. Launch/Relaunch App then test with PostMan

Get record by EmployeeID

17. Modify the EmployeeService.java to add a method that handles getting record by id.

EmployeeService.java

```
package com.example.demo.service;

import java.util.List;

import com.example.demo.model.Employee;

public interface EmployeeService {
        Employee saveEmployee(Employee employee);
        List<Employee> getAllEmployees();
```

```
        Employee getEmployeeById(long id);

}
```

18. Modify the EmployeeServiceImpl to implement the service methods

EmployeeServiceImpl.java

```java
package com.example.demo.service.impl;

import java.util.List;
import java.util.Optional;
import org.springframework.stereotype.Service;

import com.example.demo.exception.ResourceNotFoundException;
import com.example.demo.model.Employee;
import com.example.demo.repository.EmployeeRepository;
import com.example.demo.service.EmployeeService;

@Service
public class EmployeeServiceImpl implements EmployeeService {

        // dependency injection from the repository class
        private EmployeeRepository employeeRepository;

        // create constructor for this class
        public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
                super();
                this.employeeRepository = employeeRepository;
        }

        @Override
        public Employee saveEmployee(Employee employee) {
                return employeeRepository.save(employee);
        }

        @Override
        public List<Employee> getAllEmployees() {
                // the code inside this method had been manually modified
                return employeeRepository.findAll();
        }

        @Override
        public Employee getEmployeeById(long id) {
//              Optional<Employee> employee = employeeRepository.findById(id);
//              if(employee.isPresent()) {
//                      return employee.get();
//              }else {
```

```
//              throw new ResourceNotFoundException("Employee", "Id", id);
//          }

            // or shorter by using lambda
            return employeeRepository.findById(id).orElseThrow(
                    () -> new ResourceNotFoundException("Employee", "Id", id)

                    );

    }

    @Override
    public Employee updateEmployee(Employee employee, long id) {
            // TODO Auto-generated method stub
            return null;
    }
}
```

19. Modify EmployeesController.java to add a REST API endpoint to call the findbyid method in our service

EmployeesController.java

```
package com.example.demo.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.model.Employee;
import com.example.demo.service.EmployeeService;
import com.example.demo.service.impl.EmployeeServiceImpl;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    private EmployeeService employeeService;

    public EmployeeController(EmployeeService employeeService) {
            super();
```

```
            this.employeeService = employeeService;
        }

        @GetMapping("/test")
        public String test() {
                return "controller ok";
        }

        // API Endpoints for CRUD operations
        // save new record
        @PostMapping("/new")
        public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee){
                return new ResponseEntity<Employee>(
                                employeeService.saveEmployee(employee),
                                HttpStatus.CREATED
                                );
        }

        // get all employees
        @GetMapping
        public List<Employee> getAllEmployee(){
                return employeeService.getAllEmployees();
        }

        // get employee by id
        @GetMapping("{id}")
        public ResponseEntity<Employee> getEmployeeById(@PathVariable("id") long employeeId){
                return new
ResponseEntity<Employee>(employeeService.getEmployeeById(employeeId),HttpStatus.OK);
        }
}
```

20. Launch/Relaunch the application and test with Postman

| GET | ∨ | http://localhost:8080/api/employees/4 |

Params   Authorization   Headers (6)   **Body**   Pre-request Script   Tests   Settings

⦿ none   ○ form-data   ○ x-www-form-urlencoded   ○ raw   ○ binary   ○ GraphQL

This request does not have a body

Body   Cookies   Headers (5)   Test Results   ⊕   Status: 200 OK

Pretty   Raw   Preview   Visualize

{"id":4,"firstName":"John","lastName":"Goh","email":"jg@abc.com"}

21. Modify the EmployeeService.java to add method that handles updating of record

```
package com.example.demo.service;

import java.util.List;
import com.example.demo.model.Employee;

public interface EmployeeService {
        Employee saveEmployee(Employee employee);
        List<Employee> getAllEmployees();
        Employee getEmployeeById(long id);
        Employee updateEmployee(Employee employee, long id);
}
```

22. Modify the EmployeeServiceImpl.java to add method that implements the new method in the service class

```
package com.example.demo.service.impl;

import java.util.List;
import java.util.Optional;
import org.springframework.stereotype.Service;
import com.example.demo.exception.ResourceNotFoundException;
import com.example.demo.model.Employee;
import com.example.demo.repository.EmployeeRepository;
import com.example.demo.service.EmployeeService;

@Service
public class EmployeeServiceImpl implements EmployeeService {

        // dependency injection from the repository class
        private EmployeeRepository employeeRepository;

        // create constructor for this class
        public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
                super();
                this.employeeRepository = employeeRepository;
        }

        @Override
        public Employee saveEmployee(Employee employee) {
                return employeeRepository.save(employee);
        }

        @Override
```

```java
        public List<Employee> getAllEmployees() {
                // the code inside this method had been manually modified
                return employeeRepository.findAll();
        }

        @Override
        public Employee getEmployeeById(long id) {
//              Optional<Employee> employee = employeeRepository.findById(id);
//              if(employee.isPresent()) {
//                      return employee.get();
//              }else {
//                      throw new ResourceNotFoundException("Employee", "Id", id);
//              }

                // or shorter by using lambda
                return employeeRepository.findById(id).orElseThrow(
                                () -> new ResourceNotFoundException("Employee", "Id", id)

                                );
        }

        @Override
        public Employee updateEmployee(Employee employee, long id) {
                // check if record with the id exists
                Employee existingEmployee = employeeRepository.findById(id).orElseThrow(
                                () -> new ResourceNotFoundException("Employee", "Id", id)

                                );

                existingEmployee.setFirstName(employee.getFirstName());
                existingEmployee.setLastName(employee.getLastName());
                existingEmployee.setEmail(employee.getEmail());

                // save updated employee to database
                employeeRepository.save(employee);
                return existingEmployee;
        }
}
```

23. Modify the EmployeeController.java to add the RESTful API endpoint for the update record operation.

EmployeeController.java

```java
package com.example.demo.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.model.Employee;
import com.example.demo.service.EmployeeService;
import com.example.demo.service.impl.EmployeeServiceImpl;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    private EmployeeService employeeService;

    public EmployeeController(EmployeeService employeeService) {
        super();
        this.employeeService = employeeService;
    }

    @GetMapping("/test")
    public String test() {
        return "controller ok";
    }

    // API Endpoints for CRUD operations
    // save new record
    @PostMapping("/new")
    public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee){
        return new ResponseEntity<Employee>(
                    employeeService.saveEmployee(employee),
                    HttpStatus.CREATED
                    );
    }

    // get all employees
    @GetMapping
    public List<Employee> getAllEmployee(){
        return employeeService.getAllEmployees();
    }

    // get employee by id
```

```java
    @GetMapping("{id}")
    public ResponseEntity<Employee> getEmployeeById(@PathVariable("id") long employeeId){
            return new
ResponseEntity<Employee>(employeeService.getEmployeeById(employeeId),HttpStatus.OK);
    }

    // update record
    @PutMapping("{id}")
    public ResponseEntity<Employee> updateEmployee(
                @PathVariable("id") long id,
                @RequestBody Employee employee
                )
    {
            return new ResponseEntity<Employee>(
                        employeeService.updateEmployee(employee,id),
                        HttpStatus.OK);
    }
}
```

24. Launch/Relaunch application and test with Postman

| PUT | ∨ | http://localhost:8080/api/employees/5 |

Params   Authorization   Headers (8)   Body ●   Pre-request Script   Tests   Settings

○ none   ○ form-data   ○ x-www-form-urlencoded   ● raw   ○ binary   ○ GraphQL   JSON ∨

```
1    {
2        "firstName": "Angel",
3        "lastName": "Goh",
4        "email": "ag@abc.com"
5    }
```

Body   Cookies   Headers (5)   Test Results   ⊕   Status: 200 OK

Pretty   Raw   Preview   Visualize   JSON ∨   ⇄

```
2        "id": 5,
3        "firstName": "Angel",
4        "lastName": "Goh",
5        "email": "ag@abc.com"
6    }
```

25. Modify the EmployeeService.java to add method that will delete record by id.

EmployeeService.java

```java
package com.example.demo.service;

import java.util.List;
import com.example.demo.model.Employee;

public interface EmployeeService {
        Employee saveEmployee(Employee employee);
        List<Employee> getAllEmployees();
        Employee getEmployeeById(long id);
        Employee updateEmployee(Employee employee, long id);
        void deleteEmployee(long id);
}
```

26. Modify the EmployeeServiceImpl.java to implement method of the service class

EmployeeServiceImpl.java

```java
package com.example.demo.service.impl;
import java.util.List;
import java.util.Optional;
import org.springframework.stereotype.Service;
import com.example.demo.exception.ResourceNotFoundException;
import com.example.demo.model.Employee;
import com.example.demo.repository.EmployeeRepository;
import com.example.demo.service.EmployeeService;

@Service
public class EmployeeServiceImpl implements EmployeeService {

        // dependency injection from the repository class
        private EmployeeRepository employeeRepository;

        // create constructor for this class
        public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
                super();
                this.employeeRepository = employeeRepository;
        }

        @Override
        public Employee saveEmployee(Employee employee) {
                return employeeRepository.save(employee);
        }
```

```java
        @Override
        public List<Employee> getAllEmployees() {
                // the code inside this method had been manually modified
                return employeeRepository.findAll();
        }

        @Override
        public Employee getEmployeeById(long id) {
//              Optional<Employee> employee = employeeRepository.findById(id);
//              if(employee.isPresent()) {
//                      return employee.get();
//              } else {
//                      throw new ResourceNotFoundException("Employee", "Id", id);
//              }

                // or shorter by using lambda
                return employeeRepository.findById(id).orElseThrow(
                                () -> new ResourceNotFoundException("Employee", "Id", id)

                                );
        }

        @Override
        public Employee updateEmployee(Employee employee, long id) {
                // check if record with the id exists
                Employee existingEmployee = employeeRepository.findById(id).orElseThrow(
                                () -> new ResourceNotFoundException("Employee", "Id", id)

                                );

                existingEmployee.setFirstName(employee.getFirstName());
                existingEmployee.setLastName(employee.getLastName());
                existingEmployee.setEmail(employee.getEmail());

                // save updated employee to database
                employeeRepository.save(employee);
                return existingEmployee;
        }

        @Override
        public void deleteEmployee(long id) {
                //check if record is existing before deleting
                employeeRepository.findById(id).orElseThrow(
                                () -> new ResourceNotFoundException("Employee", "Id", id)

                                );
```

```
            employeeRepository.deleteById(id);
        }
}
```

27. Modify the EmployeeController.java to add a RESTful API endpoint

```
package com.example.demo.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.model.Employee;
import com.example.demo.service.EmployeeService;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

        private EmployeeService employeeService;

        public EmployeeController(EmployeeService employeeService) {
                super();
                this.employeeService = employeeService;
        }

        @GetMapping("/test")
        // (GET) http://localhost:8080/api/employees/test
        public String test() {
                return "controller ok";
        }

        // API Endpoints for CRUD operations
        // save new record
        // (POST) http://localhost:8080/api/employees/new
        @PostMapping("/new")
        public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee){
                return new ResponseEntity<Employee>(
                                employeeService.saveEmployee(employee),
```

```java
                        HttpStatus.CREATED
                        );
        }


        // get all employees
        // http://localhost:8080/api/employees
        @GetMapping
        public List<Employee> getAllEmployee(){
                return employeeService.getAllEmployees();
        }


        // get employee by id
        // (GET) http://localhost:8080/api/employees/1
        @GetMapping("{id}")
        public ResponseEntity<Employee> getEmployeeById(@PathVariable("id") long employeeId){
                return new
ResponseEntity<Employee>(employeeService.getEmployeeById(employeeId),HttpStatus.OK);
        }


        // update record
        // (PUT) http://localhost:8080/api/employees/1
        @PutMapping("{id}")
        public ResponseEntity<Employee> updateEmployee(
                        @PathVariable("id") long id,
                        @RequestBody Employee employee
                        )
        {
                return new ResponseEntity<Employee>(
                                employeeService.updateEmployee(employee,id),
                                HttpStatus.OK
                                );
        }


        // delete record
        // (DELETE) http://localhost:8080/api/employees/1
        @DeleteMapping("{id}")
        public ResponseEntity<String> updateEmployee(@PathVariable("id") long id)
        {
                employeeService.deleteEmployee(id);
                return new ResponseEntity<String>(
                                "Employee Deleted Successfully",
                                HttpStatus.OK
                                );
        }
}
```

28. Launch/Relaunch application and test with Postman

DELETE ⌄ | http://localhost:8080/api/employees/5

Params   Authorization   Headers (6)   **Body**   Pre-request Script   Tests   Settings

● none   ● form-data   ● x-www-form-urlencoded   ● raw   ● binary   ● GraphQL

This request does not have a body

Body   Cookies   Headers (5)   Test Results   ⊕ Status: 200 OK

| Pretty | Raw | Preview | Visualize | Text ⌄   ⇶

```
1    Employee Deleted Successfully
```

Implementing Asynchronous RESTful services

Implementing asynchronous RESTful services in a web application allows you to handle long-running or resource-intensive tasks more efficiently. By using asynchronous processing, the server can start a task and then immediately return a response to the client, which is not blocked while the server processes the task in the background. This is particularly useful for operations that involve significant processing time or have to wait for external resources.

<mark>**Demo: Asynchronous RESTful Services in Spring Boot**</mark>
In Spring Boot, you can achieve asynchronous behavior in your REST controllers by using Spring's asynchronous support, which involves the @Async annotation and CompletableFuture or other Future implementations. Here's how you can set it up:

    1. Enable Asynchronous Operations
First, you need to enable asynchronous operations in your Spring Boot application. This is done by adding the @EnableAsync annotation to one of your configuration classes, typically the main application class.

```
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@EnableAsync
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

2. Create Asynchronous Service

Define a service method that performs an asynchronous operation. This method should return a Future, CompletableFuture, or another asynchronous wrapper. Annotate the method with @Async to run it in a separate thread managed by Spring.

```java
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;
import java.util.concurrent.CompletableFuture;

@Service
public class AsyncService {

  @Async
  public CompletableFuture<String> performLongRunningTask() {
    // Simulate a long-running task
    try {
      Thread.sleep(10000); // 10 seconds
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    return CompletableFuture.completedFuture("Task completed!");
  }
}
```

3. Create REST Controller

Create a REST controller that uses the asynchronous service. When the endpoint is called, it will initiate the asynchronous task and immediately return a response to the client.

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.beans.factory.annotation.Autowired;
import java.util.concurrent.CompletableFuture;

@RestController
public class AsyncController {

  @Autowired
  private AsyncService asyncService;

  @GetMapping("/startAsyncTask")
  public CompletableFuture<String> startAsyncTask() {
    return asyncService.performLongRunningTask();
  }
}
```

This setup uses CompletableFuture to handle asynchronous processing. The client receives the CompletableFuture immediately, which will complete once the actual processing is done.

4. Test Your Asynchronous Service

To test the asynchronous behavior, you can use tools like Postman or cURL to make a request to the /startAsyncTask endpoint. Even though the actual task takes 10 seconds, the response should be returned immediately, indicating that the task is being processed in the background.

Securing REST Services

Securing RESTful services is crucial to prevent unauthorized access and ensure that data remains safe and intact. There are several standard practices and mechanisms that you can use to secure your REST APIs. Below, I outline a structured approach to securing REST services, particularly focusing on Spring Boot applications, which are commonly used for building RESTful services.

1. Use HTTPS

    The first step in securing a REST service is to ensure that all communications between the client and server are encrypted. This is achieved by using HTTPS instead of HTTP.

    Configure SSL/TLS: Obtain an SSL certificate from a Certificate Authority (CA) and configure your server to use this certificate, ensuring all data transmitted is encrypted.

2. Authentication

    Authentication is the process of verifying who a user is. This can be implemented using various methods:

    Basic Authentication
    Basic Authentication involves sending a username and password with each request. While simple, it should only be used over HTTPS to prevent credential interception.
    In Spring Security, you can configure basic authentication in your security configuration:

```java
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .httpBasic();
    }
}
```

JWT (JSON Web Tokens)

JWT is a popular method for securing APIs because it allows the server to verify the token's validity without needing to query a database or store session information.

Implement JWT by using libraries such as jjwt in Spring Boot:

```java
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import java.util.Date;

public String generateToken(String username) {
  return Jwts.builder()
    .setSubject(username)
    .setIssuedAt(new Date())
    .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10)) // 10 hours
    .signWith(SignatureAlgorithm.HS256, "secret")
    .compact();
}
```

3. Authorization

   Authorization involves determining if a user has the right to perform an action. Spring Security supports various ways to manage authorization:

   Role-Based Access Control (RBAC): Users are assigned roles, and access is granted based on these roles.

   Configure method-level security using @PreAuthorize or @Secured annotations to specify roles required to access methods:

```java
import org.springframework.security.access.prepost.PreAuthorize;

public class BookService {

  @PreAuthorize("hasRole('ADMIN')")
  public Book createBook(Book book) {
    return bookRepository.save(book);
  }
}
```

4. Cross-Origin Resource Sharing (CORS)

   When your API is consumed from a domain other than its own, you need to handle CORS:

   Use Spring's @CrossOrigin annotation on controllers or globally configure CORS in the security configuration to control which domains can access your API.

```java
import org.springframework.web.bind.annotation.CrossOrigin;

@CrossOrigin(origins = "http://example.com")
public class BookController {
  // Controller methods
}
```

5. Validate Input
   Always validate input to avoid SQL injection, cross-site scripting (XSS), and other malicious attacks:

   Use Spring's validation API (@Valid, @NotNull, etc.) to ensure the data your API receives is what it expects.

```
import javax.validation.Valid;

public ResponseEntity<Book> addBook(@Valid @RequestBody Book book) {
   // Saving book
}
```

6. Rate Limiting
   Prevent abuse and DoS attacks by limiting how many requests a user can make to your API in a given period of time.

   Implement rate limiting using Spring components or integrate with a third-party service or library.

7. Use Security Headers
   Security headers help protect your API from certain types of attacks like clickjacking, XSS, and other code injection attacks.

   Configure security headers in Spring Security:

```
http
   .headers()
   .frameOptions().deny()
   .xssProtection().and()
   .contentSecurityPolicy("script-src 'self'");
```

8. Logging and Monitoring

   Keep logs of access and errors to monitor for unusual activities. Tools like Spring Boot Actuator can help monitor your application's health and expose metrics.

   By combining these strategies, you can significantly enhance the security of your REST services in a Spring environment or any other type of web application platform.

Consuming REST services using the RestTemplate and Spring WebClient

Consuming REST services in a Spring application can be done using two popular methods provided by Spring Framework: RestTemplate and WebClient. RestTemplate has been a part of Spring since version 3.0, but starting from Spring 5, it is in maintenance mode and Spring recommends using the more modern WebClient, which supports synchronous and asynchronous operations and is part of the reactive stack.

Demo: Consuming REST Services with RestTemplate
   1. Add Dependencies
For a Maven project, ensure you have the Spring Boot Starter Web, which includes RestTemplate support:

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

2. Configure RestTemplate Bean

In your Spring configuration, define a RestTemplate bean. This enables you to inject RestTemplate wherever you need it in your application.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {
  @Bean
  public RestTemplate restTemplate() {
    return new RestTemplate();
  }
}
```

3. Consume a REST Service

Create a service that injects RestTemplate and uses it to call a REST API:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class UserService {

  @Autowired
  private RestTemplate restTemplate;

  public User getUserDetails(String userId) {
    String url = "https://api.example.com/users/" + userId;
    return restTemplate.getForObject(url, User.class);
  }
}
```

In this example, User is a model class that should match the JSON structure returned by the API. For instance:

```
public class User {
  private String name;
  private String email;
  // Getters and setters
}
```

WebClient is a more versatile and powerful tool for web requests, supporting both synchronous and asynchronous operations.

1. Add Dependencies

Ensure your project includes the reactive web starter, which provides WebClient:

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

2. Configure WebClient Bean

Define a WebClient bean in your Spring configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class WebClientConfig {
    @Bean
    public WebClient webClient(WebClient.Builder builder) {
        return builder
            .baseUrl("https://api.example.com")
            .build();
    }
}
```

3. Consume a REST Service

Create a service that uses WebClient to make asynchronous requests:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class UserService {

    @Autowired
    private WebClient webClient;

    public Mono<User> getUserDetails(String userId) {
        return webClient.get()
                .uri("/users/{id}", userId)
                .retrieve()
                .bodyToMono(User.class);
    }
}
```

In this example, the getUserDetails method returns a Mono<User>, which is a reactive type from Project Reactor that represents an asynchronous single or empty value. This is part of the reactive programming model that WebClient utilizes.

Demo: WebClient for RESTfull API CRUD

Setting Up WebClient

Before diving into the CRUD examples, you first need to set up a WebClient instance. This is usually done in a configuration class where WebClient is defined as a Spring bean:

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class WebClientConfig {

  @Bean
  public WebClient webClient(WebClient.Builder builder) {
    return builder.baseUrl("http://api.example.com").build();
  }
}
```

This configuration sets a base URL for all requests made through the WebClient. You can customize the builder further with more specific timeouts, headers, etc.

CRUD Operations with WebClient
  1. Create (POST)
To create a new resource (e.g., posting a new user), you can use the post method of WebClient.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class UserService {

  @Autowired
  private WebClient webClient;

  public Mono<User> createUser(User user) {
    return webClient.post()
            .uri("/users")
            .body(Mono.just(user), User.class)
            .retrieve()
            .bodyToMono(User.class);
  }
}
```

2. Read (GET)

To fetch resources (e.g., getting a user by ID), you use the get method of WebClient.

```
public Mono<User> getUserById(String id) {
   return webClient.get()
           .uri("/users/{id}", id)
           .retrieve()
           .bodyToMono(User.class);
}
```

3. Update (PUT)

To update an existing resource, you can use the put method of WebClient.

```
public Mono<User> updateUser(String id, User newUserDetails) {
   return webClient.put()
           .uri("/users/{id}", id)
           .body(Mono.just(newUserDetails), User.class)
           .retrieve()
           .bodyToMono(User.class);
}
```

4. Delete

To delete a resource (e.g., deleting a user), you use the delete method of WebClient.

```
public Mono<Void> deleteUser(String id) {
   return webClient.delete()
           .uri("/users/{id}", id)
           .retrieve()
           .bodyToMono(Void.class);
}
```

Error Handling

It's important to handle errors appropriately when working with WebClient. You can manage errors directly in the stream using methods like onErrorMap to map known exceptions to a more appropriate type or handle them in more generic error handling methods:

```
public Mono<User> getUserByIdHandlingErrors(String id) {
   return webClient.get()
           .uri("/users/{id}", id)
           .retrieve()
           .onStatus(HttpStatus::is4xxClientError, response ->
               Mono.error(new RuntimeException("Not found")))
           .bodyToMono(User.class)
           .onErrorMap(OriginalExceptionType.class, ex -> new CustomExceptionType("Custom message"));
}
```

Testing REST services using Spring Test framework

1. Set Up Testing Dependencies

Ensure your pom.xml (for Maven projects) or build.gradle (for Gradle projects) includes dependencies for Spring Boot Test and MockMvc. Here's an example setup for Maven:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
```

```
      <version>YourSpringBootVersion</version>
      <scope>test</scope>
      <exclusions>
        <exclusion>
          <groupId>org.junit.vintage</groupId>
          <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
</dependencies>
```

2. Configure MockMvc in Your Tests

Create a test class and configure MockMvc to integrate with Spring MVC. Use @WebMvcTest for testing a single controller, or @SpringBootTest along with @AutoConfigureMockMvc for full system tests that require the whole application context.

Using @WebMvcTest (for controller-specific tests):

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.boot.test.mock.mockito.MockBean;

@WebMvcTest(YourController.class)
public class YourControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private YourService yourService;  // Mock your dependencies

    // Your tests go here
}
```

Using @SpringBootTest (for broader integration tests):

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

@SpringBootTest
@AutoConfigureMockMvc
public class ApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    // Your tests go here
}
```

3. Write Tests

Now, let's write some tests. We'll test different aspects of the HTTP response, such as the status code and the body content.

Testing HTTP Status

To test that an endpoint returns the correct HTTP status:

```java
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

@Test
public void shouldReturnDefaultMessage() throws Exception {
  this.mockMvc.perform(MockMvcRequestBuilders.get("/your-endpoint"))
      .andExpect(MockMvcResultMatchers.status().isOk());  // Check for HTTP 200
}
```

Testing JSON Resources

To verify that the response body contains the expected JSON, use jsonPath:

```java
@Test
public void shouldReturnExpectedUser() throws Exception {
  this.mockMvc.perform(MockMvcRequestBuilders.get("/users/1"))
      .andExpect(MockMvcResultMatchers.status().isOk())
      .andExpect(MockMvcResultMatchers.jsonPath("$.name").value("John Doe"))
      .andExpect(MockMvcResultMatchers.jsonPath("$.email").value("john@example.com"));
}
```

Testing with Mocked Data

When you need to test the behavior of your endpoints with specific data, mock the service layer to return predetermined data:

```java
import static org.mockito.BDDMockito.given;
import org.springframework.http.MediaType;

@Test
public void shouldReturnCustomUser() throws Exception {
  User customUser = new User("CustomUser", "custom@example.com");
  given(this.yourService.findUserById(1)).willReturn(customUser);

  this.mockMvc.perform(MockMvcRequestBuilders.get("/users/1")
      .accept(MediaType.APPLICATION_JSON))
      .andExpect(MockMvcResultMatchers.status().isOk())
      .andExpect(MockMvcResultMatchers.jsonPath("$.name").value("CustomUser"));
}
```

Integrating JakartaEE JAX-RS 3.x to Spring 6

<mark>Demo: Quick Example</mark>
1. Create a new Maven Project
2. Update Project Dependencies

pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.example</groupId>
 <artifactId>jaxrsdemo</artifactId>
 <version>0.0.1-SNAPSHOT</version>

 <properties>
        <jakarta.ws.version>3.1.0</jakarta.ws.version>
        <jersey.version>3.1.6</jersey.version>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
</properties>

<dependencies>

        <dependency>
         <groupId>jakarta.ws.rs</groupId>
         <artifactId>jakarta.ws.rs-api</artifactId>
         <version>${jakarta.ws.version}</version>
        </dependency>

        <dependency>
         <groupId>org.glassfish.jersey.containers</groupId>
         <artifactId>jersey-container-jetty-http</artifactId>
         <version>${jersey.version}</version>
        </dependency>

        <dependency>
         <groupId>org.glassfish.jersey.inject</groupId>
         <artifactId>jersey-hk2</artifactId>
         <version>${jersey.version}</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
        <dependency>
          <groupId>org.projectlombok</groupId>
          <artifactId>lombok</artifactId>
          <version>1.14.8</version>
          <scope>provided</scope>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter-api</artifactId>
          <version>5.10.2</version>
          <scope>test</scope>
        </dependency>
```

```
        <dependency>
         <groupId>org.glassfish.jersey.media</groupId>
         <artifactId>jersey-media-json-jackson</artifactId>
         <scope>runtime</scope>
         <version>3.1.0</version>
        </dependency>

</dependencies>
 </project>
```

3. Create an entry point

Main.java

```java
package jaxrsdemo;
import java.net.URI;
import org.eclipse.jetty.server.Server;
import org.glassfish.jersey.jetty.JettyHttpContainerFactory;
import org.glassfish.jersey.server.ResourceConfig;
import jakarta.ws.rs.ApplicationPath;

@ApplicationPath("/")
public class Main extends ResourceConfig {

 public static final String BASE_URI = "http://localhost:8080/";

 public Main() {
   packages("jaxrsdemo");
 }

 public static void main(String[] args) {
   startServer();
 }

 public static Server startServer() {
   Server server = JettyHttpContainerFactory.createServer(URI.create("http://localhost:8080/"), new Main());
   System.out.println("Jersey application started at http://localhost:8080/");
   System.out.println("Press Ctrl+C to stop the server.");
   return server;
 }
}
```

4. Create a Data Entity

Product.java

```java
package jaxrsdemo;
import lombok.Data;

@Data
public class Product {

 long id;
```

```java
    String name;

    public Product(long id, String name) {
            super();
            this.id = id;
            this.name = name;
    }

    public Product() {
            super();
    }

    public long getId() {
            return id;
    }

    public void setId(long id) {
            this.id = id;
    }

    public String getName() {
            return name;
    }

    public void setName(String name) {
            this.name = name;
    }


}
```

5. Create a JAX-RS Resource ("Controllers" as known in spring boot). Resources is not required to have the suffix "Resource" but it is recommended.

MyResource.java

```java
package jaxrsdemo;

import java.util.ArrayList;
import java.util.List;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("/items")
public class MyResource {

 @GET
 @Produces(MediaType.APPLICATION_JSON)
 public List<Item> getAll() {
```

```
        List<Item> items = new ArrayList<Item>();
        items.add(new Item(1L,"item1"));
        items.add(new Item(2L,"item2"));
        items.add(new Item(3L,"item3"));
        return items;
    }
}
```

6. Run and test (run as a typical maven app)

<mark>Demo: Sample Jax-RS Jersey Implementation for CRUD (no persistence)</mark>
1. Create a maven project
2. Add dependencies
pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>jaxrsdemo</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
        <jakarta.ws.version>3.1.0</jakarta.ws.version>
        <jersey.version>3.1.6</jersey.version>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
</properties>

<dependencies>

        <dependency>
         <groupId>jakarta.ws.rs</groupId>
         <artifactId>jakarta.ws.rs-api</artifactId>
         <version>${jakarta.ws.version}</version>
        </dependency>

        <dependency>
         <groupId>org.glassfish.jersey.containers</groupId>
         <artifactId>jersey-container-jetty-http</artifactId>
         <version>${jersey.version}</version>
        </dependency>

        <dependency>
         <groupId>org.glassfish.jersey.inject</groupId>
         <artifactId>jersey-hk2</artifactId>
         <version>${jersey.version}</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
```

```xml
        <dependency>
          <groupId>org.projectlombok</groupId>
          <artifactId>lombok</artifactId>
          <version>1.14.8</version>
          <scope>provided</scope>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter-api</artifactId>
          <version>5.10.2</version>
          <scope>test</scope>
        </dependency>

        <dependency>
         <groupId>org.glassfish.jersey.media</groupId>
         <artifactId>jersey-media-json-jackson</artifactId>
         <scope>runtime</scope>
         <version>3.1.0</version>
        </dependency>

</dependencies>
 </project>
```

3. Create model class

Product.java

```java
...
public class Product {
   private int id;
   private String name;
   private float price;

   public Product(int id) {
      this.id = id;
   }

   public Product() {
   }

   public Product(int id, String name, float price) {
      this.id = id;
      this.name = name;
      this.price = price;
   }

      // getters and setters are not shown for brevity

   @Override
   public int hashCode() {
```

```java
      final int prime = 31;
      int result = 1;
      result = prime * result + id;
      return result;
   }

   @Override
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (getClass() != obj.getClass())
         return false;
      Product other = (Product) obj;
      if (id != other.id)
         return false;
      return true;
   }
}
...
```

4. Create a DAO class to hold CRUD Operation logic
ProductDAO.java

```java
...

public class ProductDAO {
   private static ProductDAO instance;
   private static List<Product> data = new ArrayList<>();

   static {
      data.add(new Product(1, "iPhone X", 999.99f));
      data.add(new Product(2, "XBOX 360", 329.50f));
   }

   private ProductDAO() {
   }

   public static ProductDAO getInstance() {
      if (instance == null) {
         instance = new ProductDAO();
      }

      return instance;
   }

   public List<Product> listAll() {
      return new ArrayList<Product>(data);
   }
```

```java
  public int add(Product product) {
    int newId = data.size() + 1;
    product.setId(newId);
    data.add(product);

    return newId;
  }

  public Product get(int id) {
    Product productToFind = new Product(id);
    int index = data.indexOf(productToFind);
    if (index >= 0) {
      return data.get(index);
    }
    return null;
  }

  public boolean delete(int id) {
    Product productToFind = new Product(id);
    int index = data.indexOf(productToFind);
    if (index >= 0) {
      data.remove(index);
      return true;
    }

    return false;
  }

  public boolean update(Product product) {
    int index = data.indexOf(product);
    if (index >= 0) {
      data.set(index, product);
      return true;
    }
    return false;
  }
}
```

5. Create a Jax-RS Resource
ProductResource.java

```java
...
@Path("/products")
public class ProductResource {
  private ProductDAO dao = ProductDAO.getInstance();

@GET
@Produces(MediaType.APPLICATION_JSON)
public Response list() {
  List<Product> listProducts = dao.listAll();
```

```java
    if (listProducts.isEmpty()) {
      return Response.noContent().build();
    }

    return Response.ok(listProducts).build();
  }

  @GET
  @Path("{id}")
  @Produces(MediaType.APPLICATION_JSON)
  public Response get(@PathParam("id") int id) {
    Product product = dao.get(id);
    if (product != null) {
      return Response.ok(product, MediaType.APPLICATION_JSON).build();
    } else {
      return Response.status(Response.Status.NOT_FOUND).build();
    }
  }

  @POST
  @Consumes(MediaType.APPLICATION_JSON)
  public Response add(Product product) throws URISyntaxException {
    int newProductId = dao.add(product);
    URI uri = new URI("/products/" + newProductId);
    return Response.created(uri).build();
  }

  @PUT
  @Consumes(MediaType.APPLICATION_JSON)
  @Path("{id}")
  public Response update(@PathParam("id") int id, Product product) {
    product.setId(id);
    if (dao.update(product)) {
      return Response.ok().build();
    } else {
      return Response.notModified().build();
    }
  }

  @DELETE
  @Path("{id}")
  public Response delete(@PathParam("id") int id) {
    if (dao.delete(id)) {
      return Response.noContent().build();
    } else {
      return Response.notModified().build();
    }
  }

}
```

Integrating Jax-RS Jersey in a Spring Boot Application

1. Setup Your Project Environment

Maven Dependencies
Add the following dependencies in your pom.xml file for Jersey (a popular JAX-RS implementation) and Spring:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>3.2.5</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.psatraining</groupId>
        <artifactId>springjax</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>springjax</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>17</java.version>
                <jakarta.ws.version>3.1.0</jakarta.ws.version>
                <jersey.version>3.1.6</jersey.version>

        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter</artifactId>
                </dependency>

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

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>

                        <dependency>
                        <groupId>jakarta.ws.rs</groupId>
                        <artifactId>jakarta.ws.rs-api</artifactId>
                        <version>${jakarta.ws.version}</version>
                </dependency>
```

```xml
        <dependency>
                <groupId>org.glassfish.jersey.inject</groupId>
                <artifactId>jersey-hk2</artifactId>
                <version>${jersey.version}</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
        <dependency>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>1.14.8</version>
                <scope>provided</scope>
        </dependency>

        <dependency>
                <groupId>org.glassfish.jersey.media</groupId>
                <artifactId>jersey-media-json-jackson</artifactId>
                <scope>runtime</scope>
                <version>3.1.0</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-jersey -
->

        <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-jersey</artifactId>
          <version>3.2.5</version>
        </dependency>


        <dependency>
                <groupId>org.glassfish.jersey.containers</groupId>
                <artifactId>jersey-container-jetty-http</artifactId>
                <version>${jersey.version}</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
        </plugins>
    </build>
</project>
```

2. Spring Application Class

Set up your main Spring Boot application class.

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

@SpringBootApplication
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

3. Setup your spring boot @RestController

Spring1Controller.java

```java
package com.example.demo;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class Spring1Controller {

        @GetMapping("/test1")
        public String springtest1() {
                return "Spring Test 1";
        }

        @GetMapping("/test2")
        public String springtest2() {
                return "Spring Test 2";
        }
}
```

4. Configure JAX-RS in Spring

Create a configuration class to initialize JAX-RS with Jersey, the @Component annotation allows this JaxRS entrypoint to be scanned by the spring boot entrypoint

JerseyConfig.java

```java
import org.glassfish.jersey.server.ResourceConfig;
import org.springframework.stereotype.Component;

@Component
public class JerseyConfig extends ResourceConfig {
  public JerseyConfig() {
    // Register JAX-RS application components
    register(UserResource.class);
    property("jersey.config.server.provider.packages", "com.example.demo");
  }
}
```

5. If using spring mvc alongside jaxrs, set the path that the spring mvc will use
applilcation.properties

```
spring.application.name=springjax
server.port=8080

spring.mvc.servlet.path=/spring
```

6. Test Spring boot application endpoints

```
http://localhost:8080/spring/test1
http://localhost:8080/spring/test2
```

7. Create a JaxRS Resource to test endpoint accessibility
JaxrsResource.java

```
package com.example.demo;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

@Path("/jax")
public class UserResource {

        @GET
        @Path("/test1")
        public String jax1test1() {
                return "i am from jax1 test1";
        }

        @GET
        @Path("/test2")
        public String jax1test2() {
                return "i am from jax1 test2";
        }


}
```

8. Test JaxRS application endpoints

```
http://localhost:8080/jax/test1
http://localhost:8080/jax/test2
```

9. Define the JAX-RS Resource Class
Create a resource class where you will define methods to respond to HTTP requests using JAX-RS annotations.

JAX-RS Resource Class

```
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.PUT;
```

```java
import jakarta.ws.rs.DELETE;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

@Path("/users")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class UserResource {

  @GET
  public Response getAllUsers() {
    // Logic to fetch all users
    return Response.ok().entity("Fetching all users").build();
  }

  @POST
  public Response createUser(String user) {
    // Logic to create a new user
    return Response.status(Response.Status.CREATED).entity("User created").build();
  }

  @PUT
  @Path("/{id}")
  public Response updateUser(@PathParam("id") String id, String user) {
    // Logic to update an existing user
    return Response.ok().entity("User updated").build();
  }

  @DELETE
  @Path("/{id}")
  public Response deleteUser(@PathParam("id") String id) {
    // Logic to delete a user
    return Response.ok().entity("User deleted").build();
  }
}
```

10. Run and Test Your Application

Run your Spring application. Your JAX-RS endpoints should now be integrated into the Spring application context and respond to HTTP requests according to their respective annotations.

Testing HTTP Methods

Use tools like Postman or cURL to test each of the endpoints:

- GET /users
- POST /users
- PUT /users/{id}
- DELETE /users/{id}

Creating the request matching

Understanding JAX-RS Request Matching
JAX-RS uses annotations to determine how requests are routed to resource methods. Here's a breakdown of the process:
    1.   URI Matching
First, JAX-RS matches the URI of the incoming request to the URIs defined by @Path annotations on the resource classes and methods.

- Class Level @Path: This sets the base URI for all resources defined in the class.
- Method Level @Path: This specifies the URI relative to the class-level path. If no class-level path is specified, the method-level path is relative to the application root.

For example:

```
@Path("/users")
public class UserResource {
  @GET
  @Path("/{id}")
  public Response getUserById(@PathParam("id") String id) {
    // Retrieve and return the user by ID
  }

  @POST
  public Response createUser(User user) {
    // Create a new user
  }
}
```

In this example, GET /users/123 would be routed to getUserById, and POST /users would go to createUser.

    2.   HTTP Method Matching
After a URI match is established, JAX-RS filters the candidates by the HTTP method annotation (@GET, @POST, @PUT, @DELETE, etc.). Each Java method intended to handle requests is annotated with an HTTP method that indicates which request type it can handle.

    3.   Media Type Matching
JAX-RS uses @Consumes and @Produces annotations to further narrow down the method based on the Content-Type and Accept HTTP headers of the request.

- @Consumes: Determines what media type the method can accept in the request body.
- @Produces: Specifies the media type that the method can send back in the response.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response updateUser(User user) {
  // Update user and return some information
}
```

In this setup, the method updateUser would only match POST requests that have a Content-Type of application/json and request a response that matches application/json.

4. Quality Factor Matching

When multiple methods can handle a request (for instance, based on the Accept header, multiple @Produces types could be applicable), JAX-RS uses quality factors (specified as parameters in the Accept header) to determine the best match.

Exception Handling in Matching

If no methods match an incoming request, JAX-RS automatically handles these cases by returning appropriate HTTP status codes:

- 404 Not Found: If no resource method matches the URI.
- 405 Method Not Allowed: If the URI matches but the HTTP method does not.
- 406 Not Acceptable: If the URI and method match but the Accept header cannot be satisfied.
- 415 Unsupported Media Type: If the Content-Type of the request body is not supported by the method.

The JAX-RS annotations (@PathParam, @QueryParam, @FormParam, @MatrixParam, @Context)

1. @PathParam

This annotation allows you to extract values from the URI path segments.

Example:

Suppose your application has a URL path that captures a user ID in the path, such as /users/{id}.

```java
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;
import jakarta.ws.rs.core.Response;

@Path("/users")
public class UserResource {

  @GET
  @Path("/{id}")
  public Response getUserById(@PathParam("id") String id) {
    return Response.ok("Fetched user with ID: " + id).build();
  }
}
```

2. @QueryParam

This annotation allows you to extract values from query parameters in the URL.

Example:

For a query parameter URL like /search?term=java, you can use @QueryParam to get the value of term.

```java
@GET
@Path("/search")
public Response search(@QueryParam("term") String searchTerm) {
  return Response.ok("Search results for: " + searchTerm).build();
}
```

3. @FormParam

This annotation is used to extract values from form submissions sent as application/x-www-form-urlencoded.

Example:
If a client submits a form with a field email, you can extract this in your method.

```java
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.FormParam;
import jakarta.ws.rs.core.Response;

@Path("/register")
public class RegisterResource {

  @POST
  public Response registerUser(@FormParam("email") String email, @FormParam("password") String password) {
    return Response.ok("User registered with email: " + email).build();
  }
}
```

4. @MatrixParam

This annotation is used to extract values from matrix parameters. Matrix parameters are URL parameters used for sending arbitrary name-value pairs in the URL path segments.

Example: For a URL /cars;make=toyota;color=blue, matrix parameters are make and color.

```java
@GET
@Path("/cars")
public Response getCarsByMakeAndColor(@MatrixParam("make") String make, @MatrixParam("color") String color) {
  return Response.ok("Cars filtered by make: " + make + " and color: " + color).build();
}
```

5. @Context

This annotation injects information about the context in which the service is executed. It can provide access to low-level HTTP details or to other context data like UriInfo, HttpHeaders, and so forth.

Example: You might want to access URI information or HTTP headers directly within your method.

```java
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.HttpHeaders;
import jakarta.ws.rs.core.UriInfo;
import jakarta.ws.rs.core.Response;

@GET
@Path("/info")
public Response getInfo(@Context UriInfo uriInfo, @Context HttpHeaders headers) {
  String path = uriInfo.getAbsolutePath().toString();
  String userAgent = headers.getRequestHeader("user-agent").get(0);
  return Response.ok("Request path: " + path + ", User-Agent: " + userAgent).build();
}
```

Integration with CXF servers

Apache CXF is an open-source services framework that helps you build and develop services using frontend programming APIs like JAX-RS and JAX-WS. It is popularly used for creating SOAP and REST web services. Integrating CXF with your applications, especially for running within a server environment, involves a few critical steps.

Setting up a JAX-RS service using CXF in a Spring environment, with a step-by-step guide and examples:
    1.   Set Up Your Maven Project
First, you need to set up a Maven project if you haven't already. Add dependencies for CXF and Spring Boot in your pom.xml. Here's how you would typically set it up:

```xml
<dependencies>
  <!-- Apache CXF -->
  <dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-spring-boot-starter-jaxrs</artifactId>
    <version>3.5.0</version>
  </dependency>

  <!-- Spring Boot Starter Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Boot Starter Tomcat -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
  </dependency>

  <!-- To package as an executable jar with embedded Tomcat -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
  </dependency>
</dependencies>
```

    2.   Create a CXF Configuration Class
You need to create a configuration class to configure the CXF servlet and set up your JAX-RS services:

CxfConfig.java

```java
import org.apache.cxf.Bus;
import org.apache.cxf.jaxrs.spring.SpringJAXRSServerFactoryBean;
import org.apache.cxf.bus.spring.SpringBus;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
```

```
public class CxfConfig {
  @Bean(destroyMethod = "shutdown")
  public SpringBus cxf() {
    return new SpringBus();
  }

  @Bean
  public SpringJAXRSServerFactoryBean cxfEndpoint() {
    SpringJAXRSServerFactoryBean endpoint = new SpringJAXRSServerFactoryBean();
    endpoint.setBus(cxf());
    endpoint.setServiceBeans(Arrays.asList(new UserService())); // Add your service beans here
    # endpoint.setAddress("/api"); // Set the base address for the services
    return endpoint;
  }
}
```

3. Implement a JAX-RS Service

Create a JAX-RS service using standard annotations. For instance, here's a simple user service:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("/cxf")
public class UserService {

        @GET
        @Path("test1")
        public String testcxf1() {
                return "iam from cxf1";
        }

        @GET
        @Path("test2")
        public String testcxf2() {
                return "iam from cxf2";
        }


        @GET
        @Path("user")
        @Produces(MediaType.APPLICATION_JSON)
        public User getUser() {
                return new User("John Doe", "john.doe@example.com");
        }
}
```

Define a User class:

```java
public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getters and Setters
}
```

4. Create the Spring Boot Application Class

Set up the main class to run the Spring Boot application:

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

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

5. Run and Test Your Application

Run your application using either your IDE or Maven command:

```
mvn spring-boot:run
```

Test your service to ensure it's working. You can use a browser or a tool like Postman to make a request to:

```
http://localhost:8080/cxf/test1
http://localhost:8080/cxf/test2
http://localhost:8080/cxf/test1
```

You should receive a JSON response with the user details.

Using the Spring Data REST

Spring Data REST is a powerful framework from the broader Spring ecosystem that builds on top of Spring Data repositories and automatically exposes them as RESTful web services. It leverages the Spring Data repository abstraction to turn your repository code into hypermedia-based HTTP resources without the need for explicitly writing controller code.

Key Features of Spring Data REST
- Automatic Endpoint Exposure: Spring Data REST automatically generates and exposes HTTP endpoints for your Spring Data repositories. This exposure includes standard CRUD (Create, Read, Update, Delete) operations on your entities.
- Hypermedia as the Engine of Application State (HATEOAS): Spring Data REST follows the HATEOAS principles, which means that responses from the server include links to other relevant parts of the API. This

feature makes it easier for clients to interact with the service dynamically, as the necessary state transitions are discoverable through these hypermedia links.

- Content Negotiation: It supports various representation formats (like JSON, XML, etc.) for data interchange, based on the Accept headers sent by the client.
- Customizable Exports: Developers can customize the paths, expose or hide specific methods, and configure how entities are represented through annotations or configuration properties.
- Search Resources: Spring Data REST can expose custom query methods as RESTful resources automatically. It also allows defining search endpoints where clients can execute predefined queries with URL parameters.
- Integration with Spring Security: It seamlessly integrates with Spring Security to manage access control to the exposed APIs.
- Pagination and Sorting: Collections exposed through Spring Data REST automatically support pagination and sorting, reducing the amount of data transferred and improving performance for large collections.

Demo: How Spring Data REST Works

Spring Data REST works by creating a layer on top of Spring Data repositories. It automatically translates calls to these repositories into appropriate web services. Here's a brief overview of how to use Spring Data REST:

1. Maven dependencies

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.5</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.psatraining</groupId>
    <artifactId>springdatarestdemo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springdatarestdemo</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>

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

```xml
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-rest -->
        <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-data-rest</artifactId>
          <version>3.2.5</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/jakarta.persistence/jakarta.persistence-api -->
        <dependency>
          <groupId>jakarta.persistence</groupId>
          <artifactId>jakarta.persistence-api</artifactId>
          <version>3.1.0</version>
        </dependency>

        <dependency>
         <groupId>org.springframework.data</groupId>
         <artifactId>spring-data-rest-webmvc</artifactId>
         <version>4.2.5</version>
        </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>

  </dependencies>
  <build>
        <plugins>
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
        </plugins>
  </build>

</project>
```

2.   Define Domain Entities

First, define your domain entities using JPA annotations:

```
package com.example.demo;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private String author;

    // constructor(s), getters and setters
```

3.   Create Repository Interfaces

Next, define a repository interface for each entity:

```
package com.example.demo;

import org.springframework.data.repository.CrudRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(collectionResourceRel = "books", path = "books")
        public interface BookRepository extends CrudRepository<Book, Long> {
}
```

4.   Configure Spring Data REST

Spring Data REST configuration is minimal, often requiring no more than the inclusion of the appropriate Spring Boot starter and some minimal application properties, if any.

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

@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

5.   Access Your RESTful API

Once the application is running, Spring Data REST exposes the CRUD operations on Book at a path like /books. You can perform standard operations such as:

```
GET /books: List all books with pagination.
POST /books: Create a new book.
GET /books/{id}: Retrieve a specific book.
```

PUT /books/{id}: Update an existing book.
DELETE /books/{id}: Delete a book.

These endpoints are created automatically and can be customized or enhanced with additional query methods, controllers, or event handlers.

<mark>Demo: Customizing endpoints in a Spring Data REST application</mark>
     1.   Change Base Path
You can change the base URI of all the repository resources globally by setting the spring.data.rest.base-path property in your application.properties or application.yml:

application.properties:

```
spring.data.rest.base-path=/api
```

This will prefix all data repository exports with /api, so instead of accessing your resources at /books, it will be /api/books.

     2.   Customize Repository Paths
You can customize the path of specific repositories by using the @RepositoryRestResource annotation on your repository interface. This allows you to specify a custom path and even change the name used for the collection resource.

For example, if you want to change the path for accessing Book entities from the default /books to /library/books:

```
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(path = "library/books", collectionResourceRel = "books")
public interface BookRepository extends PagingAndSortingRepository<Book, Long> {
}
```

- path specifies the URI at which the repository is exposed.
- collectionResourceRel specifies the name of the collection resource under the HAL document's _embedded section, which is used in HATEOAS links.

     3.   Expose/Hide Certain Repository Methods
To control the exposure of CRUD methods in your repository, you can use the @RestResource annotation on the query methods. You can either expose additional query methods or hide auto-generated ones.

For example, to prevent deletion of Book entities through the API, you can modify the repository as follows:

```
import org.springframework.data.rest.core.annotation.RestResource;

public interface BookRepository extends PagingAndSortingRepository<Book, Long> {

  @Override
  @RestResource(exported = false)
  void deleteById(Long id);

  @Override
  @RestResource(exported = false)
  void delete(Book entity);
}
```

4. Customize Exposed Data

You may want to customize the JSON output of your endpoints. For example, to exclude certain fields from the JSON representation or to expose additional fields, you can use Jackson annotations directly on your entity classes.

```java
import com.fasterxml.jackson.annotation.JsonIgnore;

@Entity
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    @JsonIgnore
    private String internalCode; // This will not be shown in JSON output

    // Getters and setters
}
```

5. Custom Controllers

If the customization provided by annotations is not sufficient, you can always add custom controllers. This is useful for adding entirely new behaviors or endpoints not directly tied to a single entity type.

For example, adding a custom endpoint to reset data or perform bulk operations:

```java
@RestController
@RequestMapping("/api/books")
public class CustomBookController {

    @PostMapping("/reset")
    public ResponseEntity<Void> resetBooks() {
        // Custom logic to reset books
        return ResponseEntity.ok().build();
    }
}
```

6. Event Handling

Spring Data REST also supports event handling, which can be used to react to or modify the behavior before or after certain repository events (like before save or after delete).

```java
@Component
@RepositoryEventHandler(Book.class)
public class BookEventHandler {

    @HandleBeforeSave
    public void handleBookSave(Book book) {
        // logic to execute before saving a book
    }
```

```
  @HandleAfterDelete
  public void handleBookAfterDelete(Book book) {
    // logic to execute after deleting a book
  }
}
```

1. Create MySQL database and user account
2. New Maven project and add dependencies

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>3.2.5</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.psatraining</groupId>
        <artifactId>springdatarestdemo</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>springdatarestdemo</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>17</java.version>
        </properties>
        <dependencies>
          <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
          </dependency>
          <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
          </dependency>
          <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
          </dependency>
                <!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
                <dependency>
                  <groupId>com.mysql</groupId>
                  <artifactId>mysql-connector-j</artifactId>
                </dependency>
```

```xml
        <dependency>
          <groupId>com.h2database</groupId>
          <artifactId>h2</artifactId>
          <scope>runtime</scope>
        </dependency>
      </dependencies>

      <build>
          <plugins>
              <plugin>
                      <groupId>org.springframework.boot</groupId>
                      <artifactId>spring-boot-maven-plugin</artifactId>
              </plugin>
          </plugins>
      </build>

</project>
```

3. Set application properties
application.properties

```
spring.application.name=springdatarestdemo
spring.datasource.url=jdbc:mysql://localhost:3306/psaspringdb1
spring.datasource.username=psaadmin
spring.datasource.password=123
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
spring.data.rest.base-path=/api
```

4. Create entity class
Book.java

```java
package com.example.demo;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Book {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String title;
  private String author;

  //constructor(s), getters and setters
}
```

5. Create repository interface
BookRepository.java

```
package com.example.demo;

import org.springframework.data.repository.CrudRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(collectionResourceRel = "books", path = "books")
public interface BookRepository extends CrudRepository<Book, Long> {
}
```

6. Test endpoints
Start your Spring Boot application. You can now access your CRUD repository via HTTP:
- GET /api/people to retrieve all people.
- POST /api/people to create a new person.
- GET /api/people/{id} to retrieve a person by ID.
- PUT /api/people/{id} to update a person.
- DELETE /api/people/{id} to delete a person.

Note:
In Spring Data, both CrudRepository and JpaRepository are interfaces that abstract the boilerplate CRUD functionality for an entity class. They are part of the Spring Data Commons and Spring Data JPA modules, respectively. Understanding the differences between these two can help you choose the right one for your needs in Spring-based applications.

CrudRepository
CrudRepository is the core repository interface in Spring Data. It provides CRUD functionality on the repository layer. It is a part of the Spring Data Commons project and can be used with any datastore by extending it and defining custom methods. Here are some of the primary methods it includes:
- save(S entity): Saves a given entity.
- findById(ID id): Retrieves an entity by its id.
- existsById(ID id): Returns whether an entity with the given id exists.
- findAll(): Returns all instances of the type.
- count(): Returns the count of entities.
- deleteById(ID id): Deletes the entity with the specified id.
- delete(S entity): Deletes a given entity.

JpaRepository
JpaRepository extends PagingAndSortingRepository, which in turn extends CrudRepository. Thus, it inherits all the functionalities of CrudRepository and PagingAndSortingRepository (like pagination and sorting), along with a few additional JPA-specific methods. It is specific to JPA and provides JPA related methods such as flushing the persistence context and delete records in a batch. Key methods include:
- findAll(Sort sort): Returns all entities sorted by the given options.
- findAll(Pageable pageable): Returns all entities according to the given paging options.
- saveAndFlush(S entity): Saves the entity and flushes changes instantly.
- deleteInBatch(Iterable<S> entities): Deletes the entities given as a batch, reducing the number of database operations.
- deleteAllInBatch(): Deletes all entities in a single batch call.

- getOne(ID id): Returns a reference to the entity with the given identifier.

Choosing Between CrudRepository and JpaRepository
- General Use vs. JPA-Specific: If you are using JPA in your application and need JPA-specific features like flushing the persistence context or batch deletion, JpaRepository is more suitable. If you are looking for a more general abstraction that could be used with different datastores (not just JPA), CrudRepository is the appropriate choice.
- Pagination and Sorting: If your application requires pagination or sorting, JpaRepository provides a straightforward approach since it extends PagingAndSortingRepository.
- Performance Considerations: JpaRepository provides some additional methods that can help optimize performance, such as deleteInBatch and deleteAllInBatch. These methods can be more efficient than using the regular delete method provided by CrudRepository.
- Simplicity vs. Functionality: CrudRepository offers simplicity and is sufficient for many basic applications that only require core CRUD functionality. However, if you need more comprehensive data access capabilities that leverage JPA, JpaRepository would be more beneficial.

Applying the Spring HATEOAS

<mark>Demo: RESTful API with full CRUD operations using Spring Boot and Spring HATEOAS.</mark>
This will involve creating a basic entity, setting up a controller with HATEOAS links, and using a data access layer.

1. Project Setup

Start by creating a new Spring Boot project. You can use the Spring Initializr (https://start.spring.io/) to generate your project. Choose Maven or Gradle as your build system and add dependencies for 'Spring Web', 'Spring Data JPA', 'Spring HATEOAS', and your choice of database connector (e.g., H2 for in-memory testing).

2. Add Dependencies

If you set up your project manually or need to add dependencies afterward, include them in your pom.xml for

Maven:

```xml
<dependencies>
  <!-- Spring Boot Starter Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Data JPA -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <!-- Spring HATEOAS -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
  </dependency>

  <!-- H2 Database -->
```

```xml
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

3. Define the Domain Model

Create a JPA entity Book:

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String author;

    // Standard getters and setters
}
```

4. Create a Repository

Use Spring Data JPA to handle data operations:

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

public interface BookRepository extends JpaRepository<Book, Long> {
}
```

5. Create an Assembler

Implement a model assembler to add HATEOAS links to your Book resources:

```java
import org.springframework.hateoas.server.mvc.RepresentationModelAssemblerSupport;
import org.springframework.stereotype.Component;

@Component
public class BookModelAssembler extends RepresentationModelAssemblerSupport<Book, BookModel> {

    public BookModelAssembler() {
        super(BookController.class, BookModel.class);
    }

    @Override
    public BookModel toModel(Book entity) {
        BookModel model = createModelWithId(entity.getId(), entity);
        model.setId(entity.getId());
        model.setTitle(entity.getTitle());
```

```
        model.setAuthor(entity.getAuthor());
        return model;
    }
}
```

Here, BookModel is a simple DTO that extends RepresentationModel:

```
import org.springframework.hateoas.RepresentationModel;

public class BookModel extends RepresentationModel<BookModel> {
    private Long id;
    private String title;
    private String author;

    // Getters and setters
}
```

6. Build the Controller

Implement CRUD operations in the controller using Spring MVC and HATEOAS:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.hateoas.EntityModel;
import org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/books")
public class BookController {

    @Autowired
    private BookRepository repository;

    @Autowired
    private BookModelAssembler assembler;

    @GetMapping("/")
    public ResponseEntity<List<EntityModel<BookModel>>> all() {
        List<EntityModel<BookModel>> books = repository.findAll().stream()
            .map(assembler::toModel)
            .collect(Collectors.toList());
        return ResponseEntity.ok(books);
    }

    @PostMapping("/")
    public ResponseEntity<EntityModel<BookModel>> newBook(@RequestBody Book newBook) {
        Book book = repository.save(newBook);
```

```
        return
ResponseEntity.created(WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder.methodOn(BookController.class).one(boo
k.getId())).toUri())
            .body(assembler.toModel(book));
    }

    @GetMapping("/{id}")
    public ResponseEntity<EntityModel<BookModel>> one(@PathVariable Long id) {
        Book book = repository.findById(id).orElseThrow(() -> new RuntimeException("Not Found"));
        return ResponseEntity.ok(assembler.toModel(book));
    }

    @PutMapping("/{id}")
    public ResponseEntity<EntityModel<BookModel>> replaceBook(@RequestBody Book newBook, @PathVariable
Long id) {
        Book updatedBook = repository.findById(id)
            .map(book -> {
                book.setTitle(newBook.getTitle());
                book.setAuthor(newBook.getAuthor());
                return repository.save(book);
            })
            .orElseGet(() -> {
                newBook.setId(id);
                return repository.save(newBook);
            });
        return ResponseEntity.ok(assembler.toModel(updatedBook));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<?> deleteBook(@PathVariable Long id) {
        repository.deleteById(id);
        return ResponseEntity.noContent().build();
    }
}
```

Step 7: Run and Test

Start your application and test each endpoint using a tool like Postman or curl. You should see that each response includes HATEOAS links that help the client navigate the API.

Setting up Swagger 2 for REST services

Swagger 2 will help you document your API and provide an interactive user interface where you can test the API.

Note:
- Swagger 2 (no longer supports spring boot 3.x)
- Swagger 2→springfox→springdoc-openapi-starter-webmvc-ui

1. Create a Spring Boot Project

Start by creating a new Spring Boot project. You can use Spring Initializr (https://start.spring.io/) to generate your project structure. Choose dependencies for 'Spring Web', 'Spring Data JPA', and your choice of database connector (e.g., H2 for an in-memory database).

2. Add Swagger Dependencies

Add the following dependencies to your pom.xml to include Swagger 2 in your project. If you're using Maven:

```
<!-- Swagger -->
  <dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
  </dependency>
```

3. Application properties

```
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger-ui.html
```

4. Access swagger ui

```
http://localhost:8080/swagger-ui.html
```

5. Manually add tags and definitions to controllers

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.tags.Tag;

@RestController
@Tag(name = "User API", description = "User management APIs")
public class UserController {

  @GetMapping("/users")
  @Operation(summary = "List all users", description = "Returns a list of users")
  public List<User> getAllUsers() {
    return userService.getAllUsers();
  }
}
```

6. Run and Access Swagger UI

Run your Spring Boot application, and visit http://localhost:8080/swagger-ui.html in your web browser. You should see the Swagger UI interface with your API endpoints listed, which you can now interact with.

Using Swagger 2 Today

While Swagger 2 is still in use, it's recommended to use the latest version of the OpenAPI Specification (currently OpenAPI 3.x) for new projects. OpenAPI 3.x introduces a number of improvements and additional features such as support for callbacks, links, and example objects, making it more comprehensive and better suited for modern API needs. However, tools like Swagger UI and Swagger Codegen continue to support both Swagger 2 and OpenAPI 3 specifications.