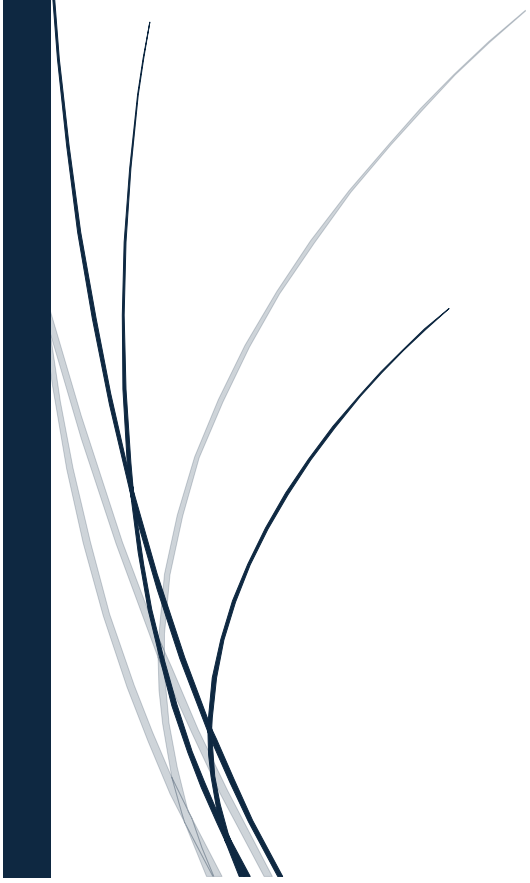
1/24/2024

Java Spring REST Services and Front-end Frameworks



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

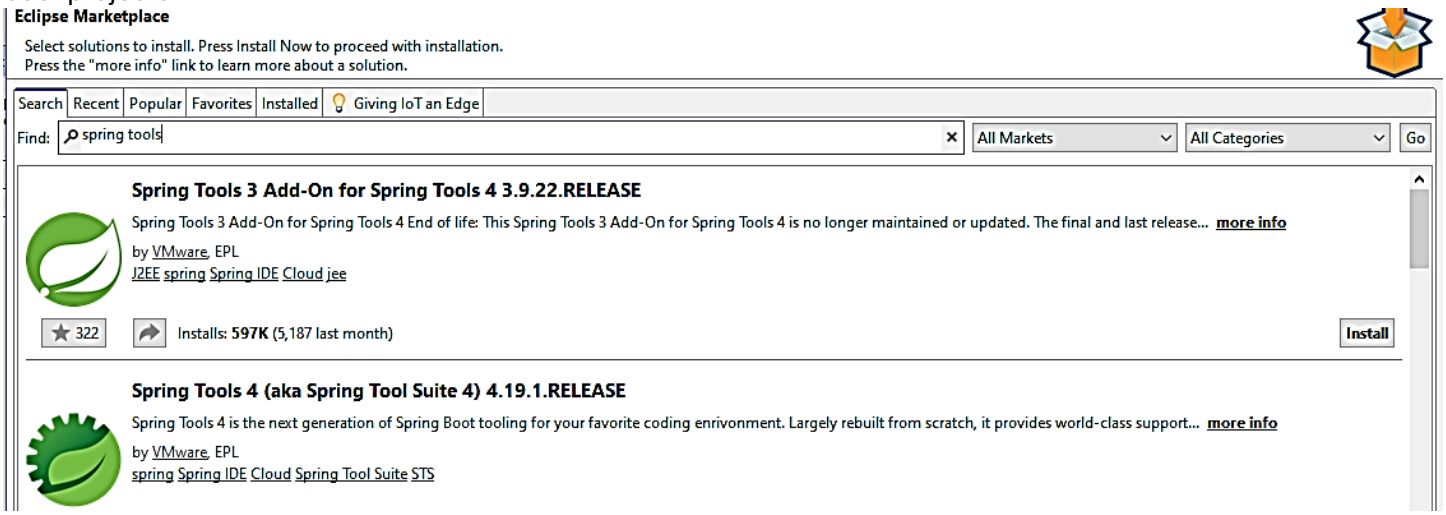
pom.xml

```
<?xml 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.



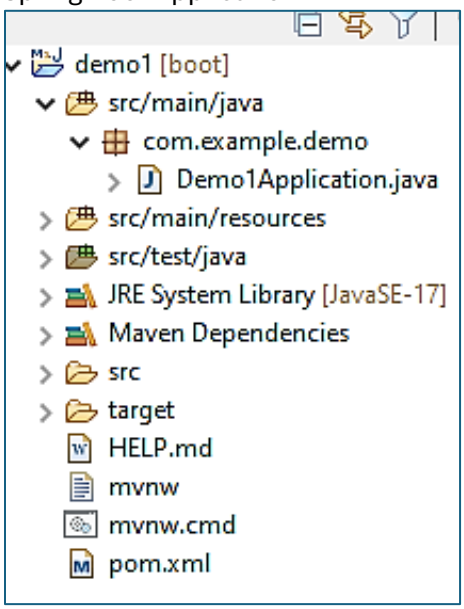
Get more information here:

https://start.spring.io/

Create Spring Boot project from Eclipse

File→New→Spring→New Spring Starter Project

Spring Boot Application



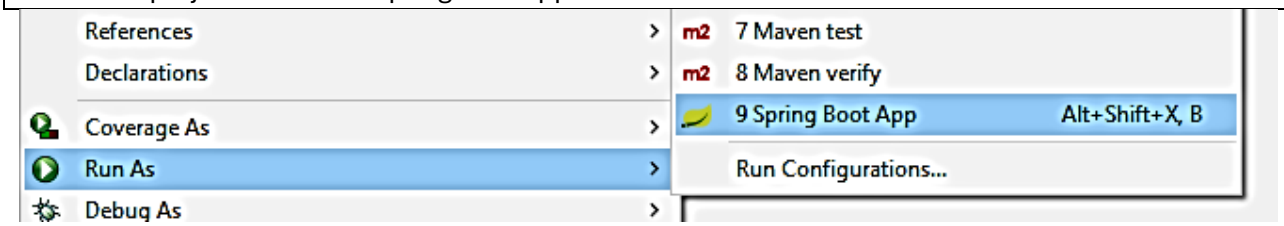

```java
package com.example.demo;

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



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

```
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

Demo: Build a RESTful API that returns JSON

1. Create a class that represents the object you need to return as a response

Employee.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:

```
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:

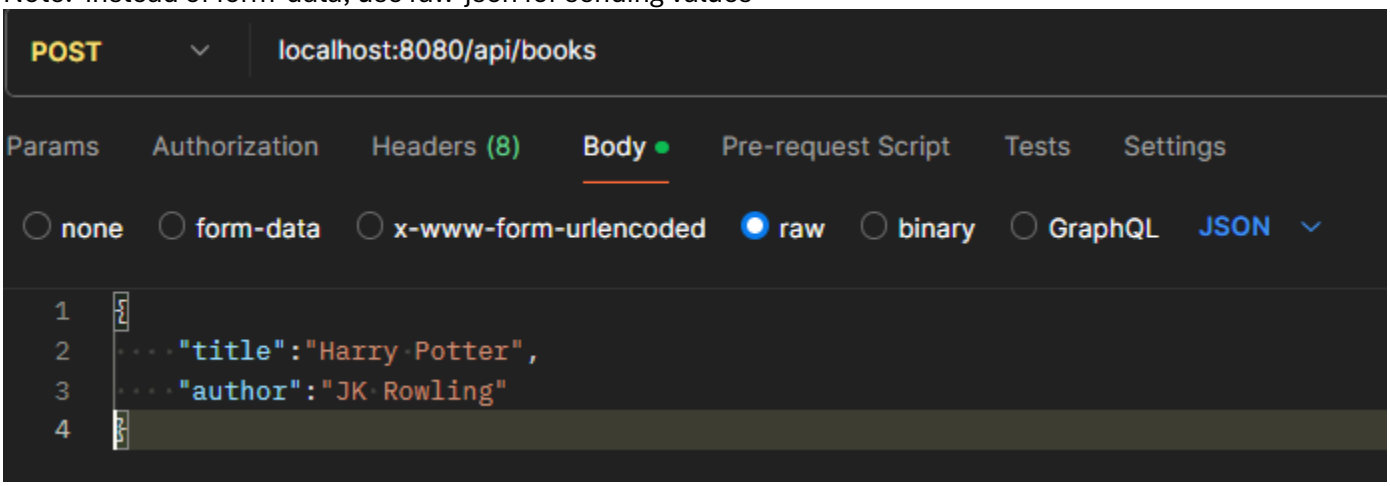
```
package com.example.bookstore.controller;

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



Use of request parameters

Build a RESTful API with Request Parameter (@RequestParam)

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

```
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

Demo: Build a RESTful API with Path Variable (@PathVariable)

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

1. Modify the EmployeeController.java and add another method

```
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.

```
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;
  }

  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.

```
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.

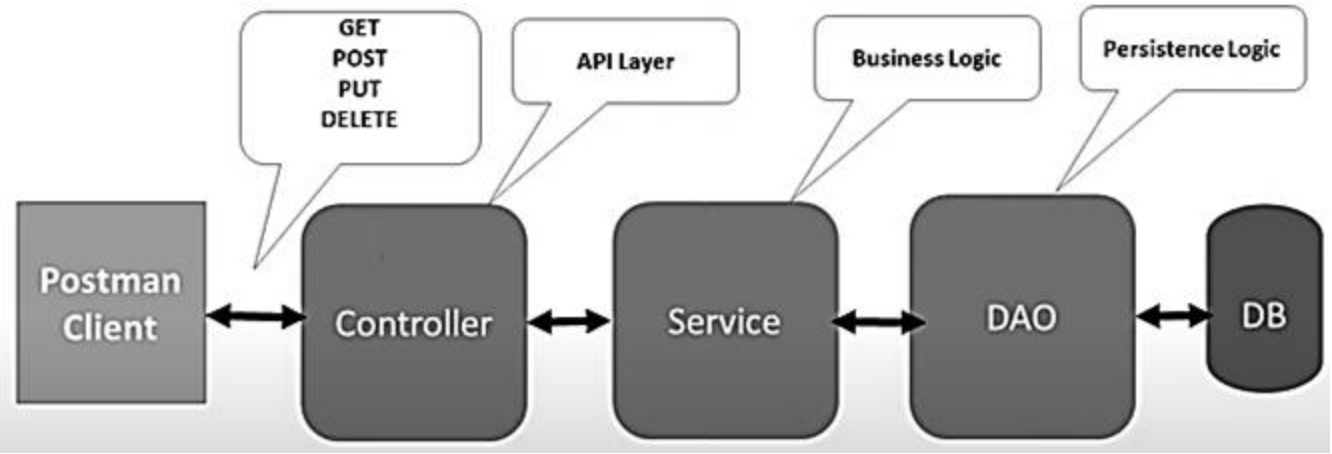
To request JSON:

```
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:

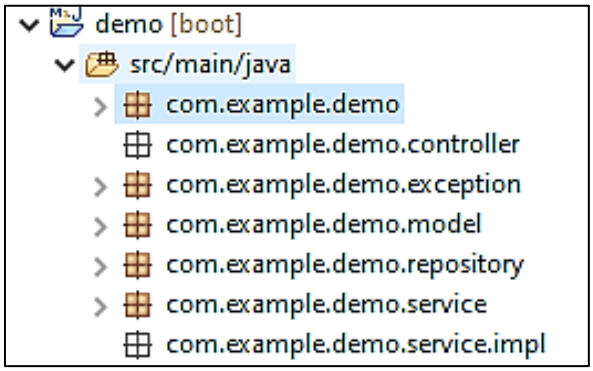
Project	Maven
Language	Java
Spring Boot	3.2.0
Group	com.restproject
Artifact	Rest1
Name	Rest1
Description	
Package Name	com.restproject.Rest1
Packaging	Jar
Java Version	20
Dependencies	Spring Web Spring Data JPA MySQL Driver Lombok

pom.xml

```
<?xml 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”)

```
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)
```

```
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

```
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);
}
```

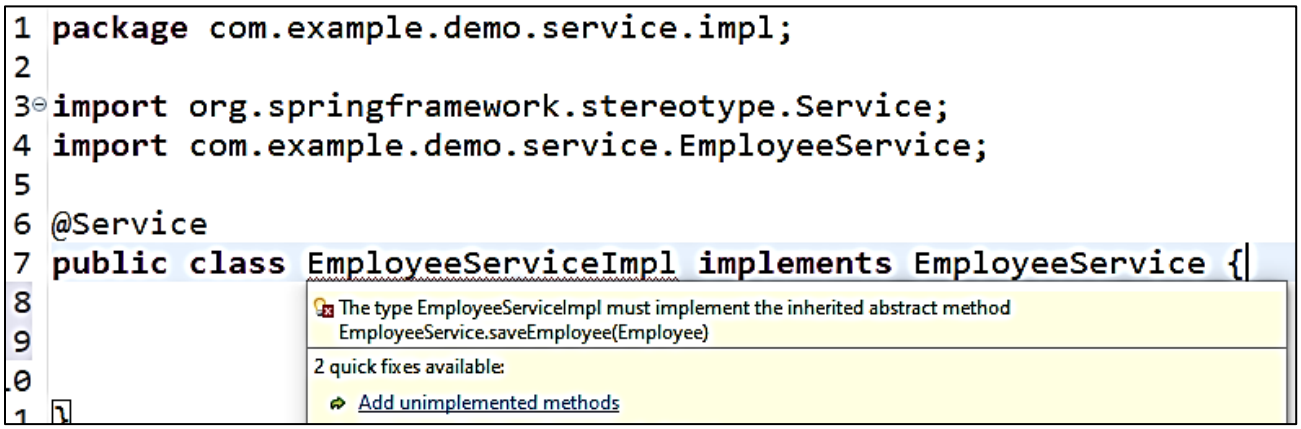
10. Create a class that implements the EmployeeService Interface

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”



Adding new records

11. Create a controller class that will provide mapping of API endpoints

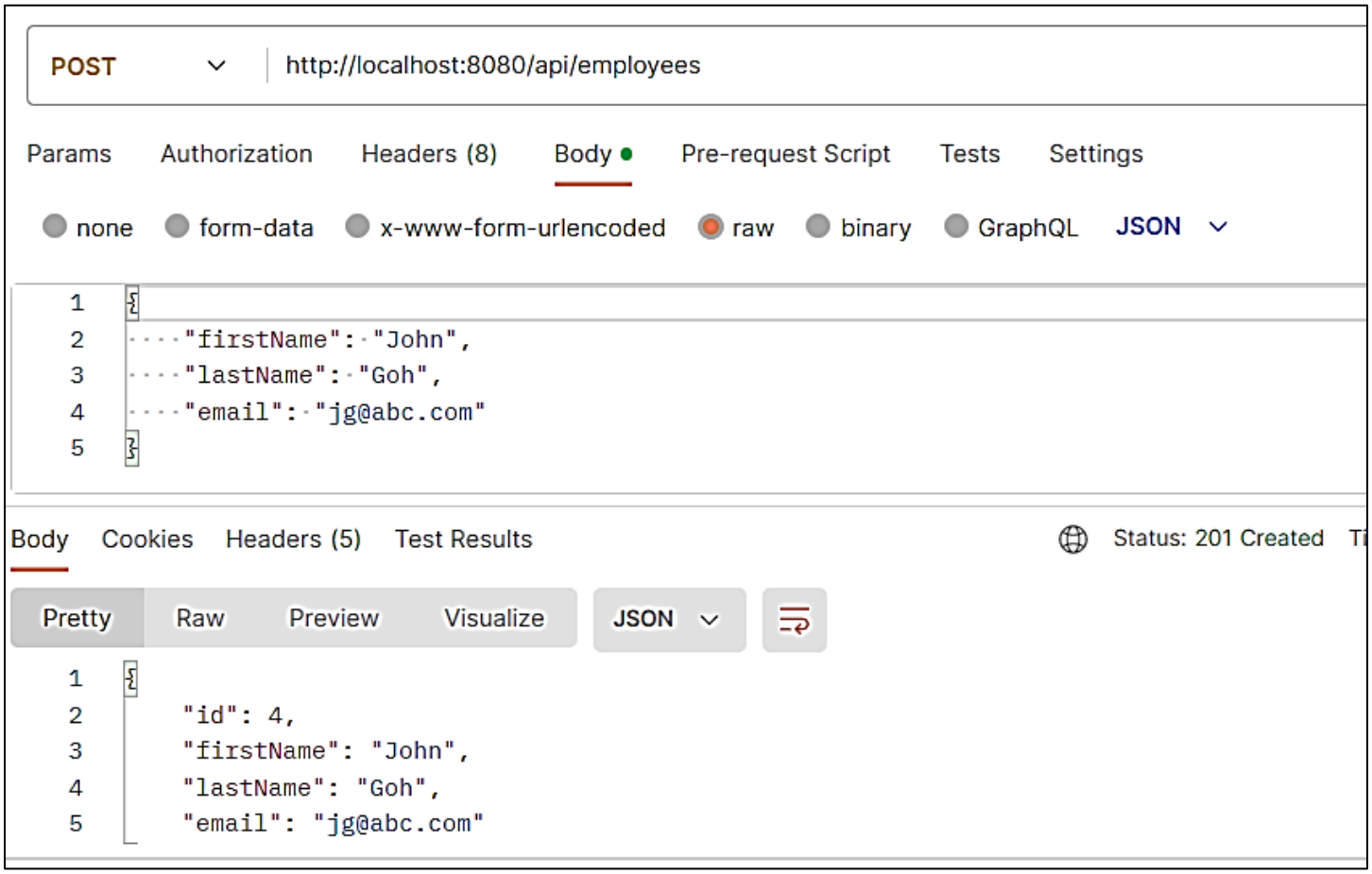
Rclick Controller package→new class (“EmployeeController”)

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



Getting all records

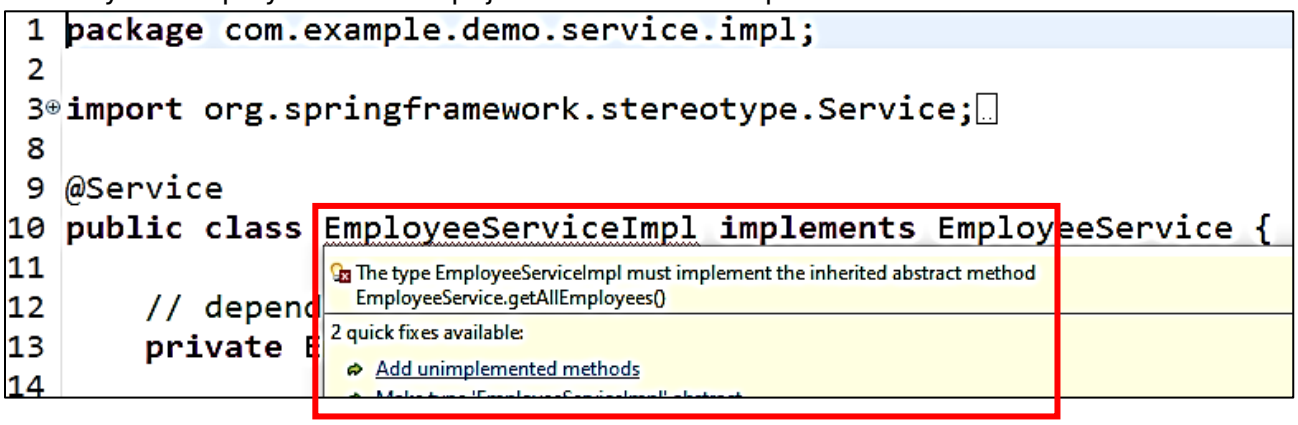
13. Modify the EmployeeService.java class to add new service method

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




```
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() {
```

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

}
```

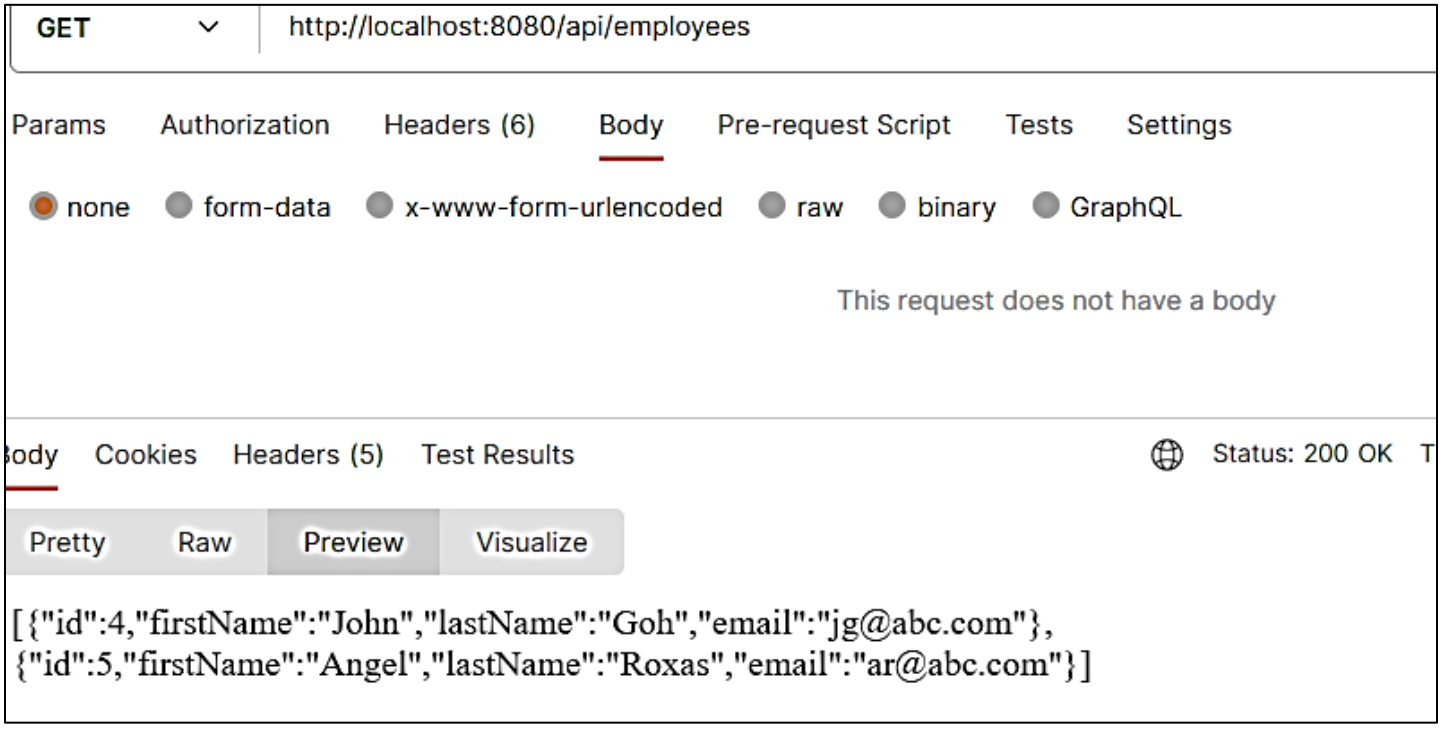
15. Modify the EmployeeController.java to add a RESTFul API Endpoint

EmployeeController.java

```
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

```
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;
        }
}
```

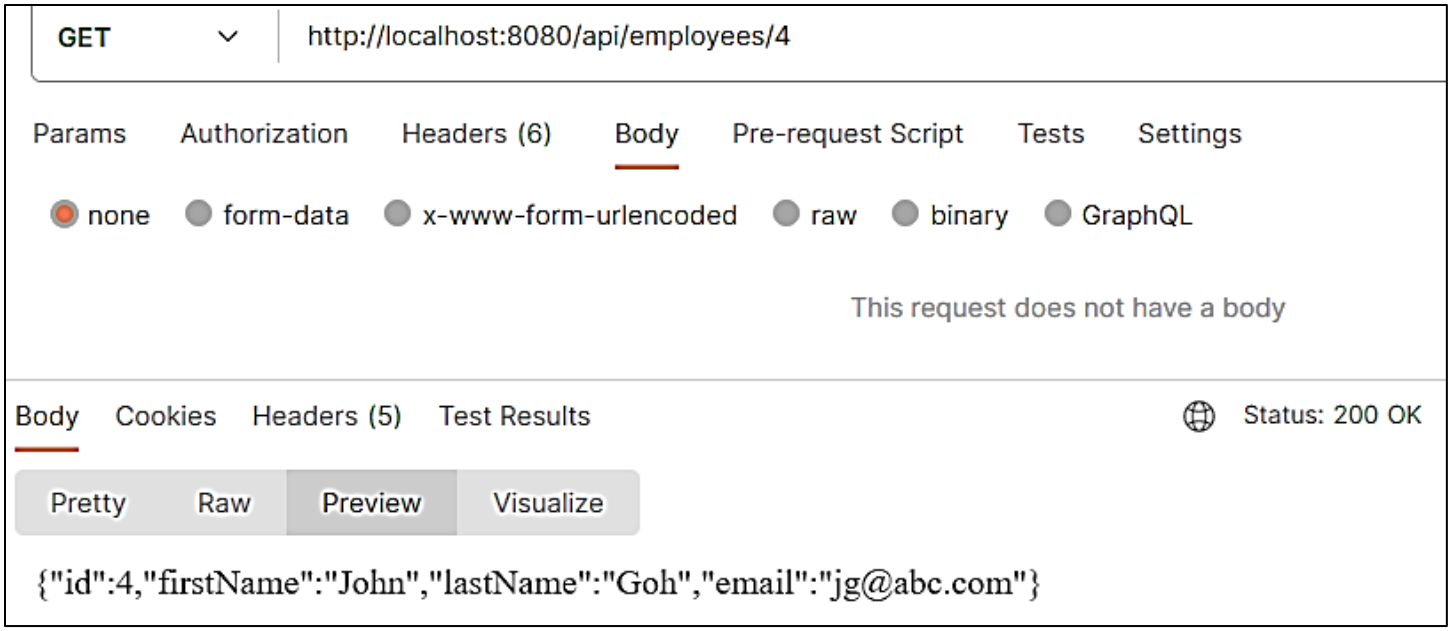
19. Modify EmployeesController.java to add a REST API endpoint to call the findbyid method in our service

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



Update record by EmployeeID

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
	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;
        }
}
```

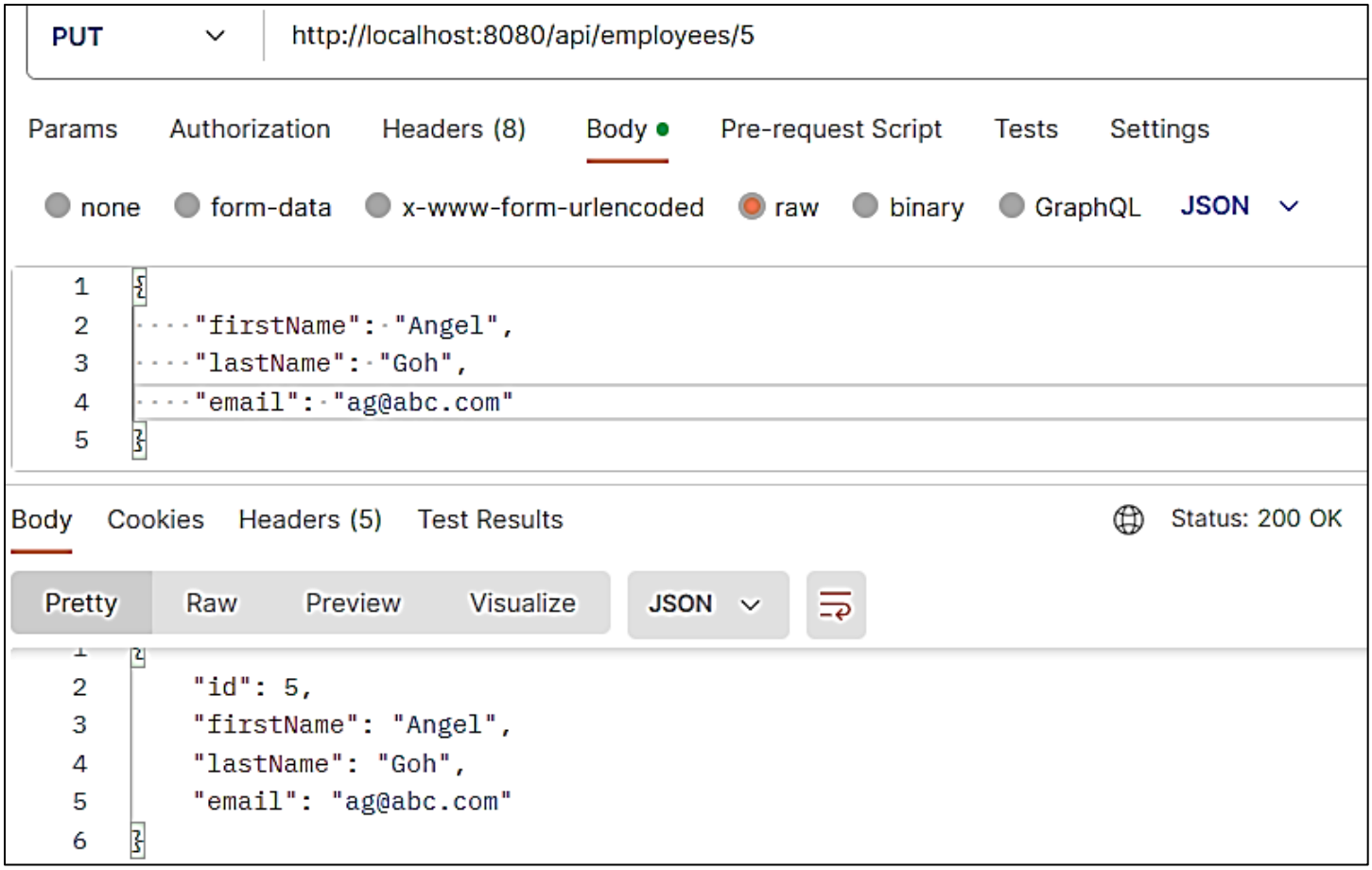
23. Modify the EmployeeController.java to add the RESTful API endpoint for the update record operation.

EmployeeController.java

```
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.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
	@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



Delete record by EmployeeID

25. Modify the EmployeeService.java to add method that will delete 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);
	Employee updateEmployee(Employee employee, long id);
	void deleteEmployee(long id);
}
```

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

EmployeeServiceImpl.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) {
		// 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);
	}
}
```

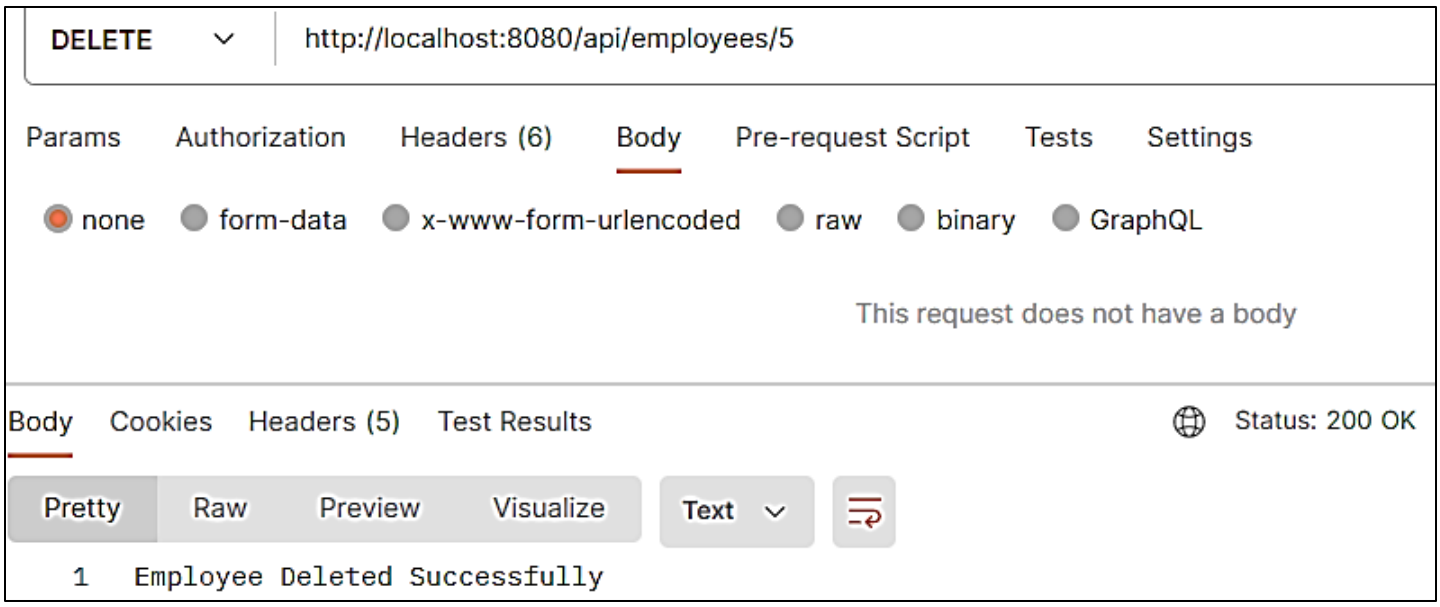
27. Modify the EmployeeController.java to add a RESTful API endpoint

```
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),
				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



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.

Demo: Asynchronous RESTful Services in Spring Boot

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.

```
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.

```
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:

```
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:

```
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:

```
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.

```
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
}
```

Demo: Consuming REST Services with WebClient

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:

```
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.

```
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):

```
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):

```
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:

```
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:

```
@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:

```
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

Demo: Quick Example

1. Create a new Maven Project
2. Update Project Dependencies

pom.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

```
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

```
package jaxrsdemo;

import lombok.Data;

@Data
public class Product {

  long id;
  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

```
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)

Demo: Sample Jax-RS Jersey Implementation for CRUD (no persistence)

1. Create a maven project
2. Add dependencies

pom.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 model class

Product.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() {
    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

```
...

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);
  }

  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

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


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

  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:

```
<dependencies>
  <!-- Spring Boot Starter Web to set up Spring and embedded Tomcat -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Jersey for JAX-RS support -->
  <dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
```

```
    <artifactId>jersey-container-servlet</artifactId>
    <version>3.0.4</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.inject</groupId>
    <artifactId>jersey-hk2</artifactId>
    <version>3.0.4</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-server</artifactId>
    <version>3.0.4</version>
  </dependency>

  <!-- Spring Boot Starter to configure Spring -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
</dependencies>
```

2. Configure JAX-RS in Spring

Create a configuration class to initialize JAX-RS with Jersey.

Jersey Configuration Class

```
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);
  }
}
```

3. 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;
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();
  }
}
```

4. Spring Application Class

Set up your main Spring Boot application class.

```
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 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}.

```
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.

```
@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.

```
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.

```
@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.

```
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:

```
<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:

```
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("/users")
public class UserService {

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public User getUser() {
```

```
    return new User("John Doe", "john.doe@example.com");
  }
}
```

Define a User class:

```
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:

```
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/api/users
```

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. Define Domain Entities

First, define your domain entities using JPA annotations:

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

@Entity
public class Book {

  @Id
  @GeneratedValue
  private Long id;
  private String title;
  private String author;

  // Getters and setters omitted for brevity
}
```

2. Create Repository Interfaces

Next, define a repository interface for each entity:

```
import org.springframework.data.repository.PagingAndSortingRepository;

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

3. 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);
  }
}
```

4. 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.

Demo: Customizing endpoints in a Spring Data REST application

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.

```
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:

```
@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).

```
@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
    }
}
```

Applying the Spring HATEOAS

Demo: RESTful API with full CRUD operations using Spring Boot and Spring HATEOAS.

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:

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

3. Define the Domain Model

Create a JPA entity Book:

```
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:

```
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:

```
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.

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 2 Dependencies

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

```
<dependencies>
  <!-- SpringFox Dependency -->
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
  </dependency>
</dependencies>
```

3. Configure Swagger 2

Create a configuration class to set up Swagger 2. This class will configure Swagger to scan your controllers and models to generate the API documentation.

```
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableSwagger2
public class SwaggerConfig {
  @Bean
  public Docket api() {
    return new Docket(DocumentationType.SWAGGER_2)
      .select()
      .apis(RequestHandlerSelectors.basePackage("com.example.yourapplication"))
      .paths(PathSelectors.any())
      .build();
  }
}
```

Replace "com.example.yourapplication" with the actual base package where your controllers are located.

4. Define Your Entity and Repository

Assuming a simple Book entity, define it along with a repository:

```
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;

  // Getters and setters
}

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

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

5. Create REST Controller with CRUD Operations

Implement a REST controller with CRUD operations. Swagger will automatically pick up the endpoints.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/books")
public class BookController {
  @Autowired
  private BookRepository bookRepository;

  @GetMapping
  public List<Book> getAllBooks() {
    return bookRepository.findAll();
  }

  @PostMapping
  public Book createBook(@RequestBody Book book) {
    return bookRepository.save(book);
  }

  @GetMapping("/{id}")
  public Book getBookById(@PathVariable Long id) {
    return bookRepository.findById(id)
        .orElseThrow(() -> new RuntimeException("Book not found on :: " + id));
  }
```

```
    @PutMapping("/{id}")
    public Book updateBook(@PathVariable Long id, @RequestBody Book bookDetails) {
        Book book = bookRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Book not found on :: " + id));

        book.setTitle(bookDetails.getTitle());
        book.setAuthor(bookDetails.getAuthor());
        return bookRepository.save(book);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<?> deleteBook(@PathVariable Long id) {
        Book book = bookRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Book not found on :: " + id));

        bookRepository.delete(book);
        return ResponseEntity.ok().build();
    }
}
```

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.

Setting up OpenAPI 3.x for REST documentation

To effectively use OpenAPI 3.x for documenting and implementing Spring REST services, you'll typically use the Springdoc OpenAPI library. This library is a popular choice for integrating OpenAPI 3.x into Spring Boot applications, allowing for automated generation of API documentation and interactive UIs.

1. Set Up Your Spring Boot Project

Create a new Spring Boot project using the Spring Initializr. Select your preferred dependencies including 'Spring Web', 'Spring Data JPA', and an appropriate database like 'H2 Database' for in-memory persistence.

2. Add Springdoc OpenAPI Dependency

Add the Springdoc OpenAPI UI library to your project to enable OpenAPI 3.x support. If you are using Maven, add this dependency to your pom.xml:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.5.9</version>
</dependency>
```

3. Configure Application Properties

Add the following properties to your application.properties or application.yml to customize the OpenAPI documentation:

properties

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

This configures the path for JSON API docs and the Swagger UI.

4. Define Your Entity and Repository

Create an entity Book and a corresponding repository. Here's an example using JPA:

```
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;

  // Getters and setters
}

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

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

5. Create a REST Controller

Implement a REST controller with CRUD operations. Add OpenAPI annotations to enrich the API documentation:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import org.springframework.http.ResponseEntity;
import java.util.List;

@RestController
@RequestMapping("/api/books")
public class BookController {
  @Autowired
  private BookRepository bookRepository;

  @GetMapping("/")
  public List<Book> getAllBooks() {
    return bookRepository.findAll();
  }

  @PostMapping("/")
  public Book createBook(@RequestBody Book book) {
    return bookRepository.save(book);
  }

  @GetMapping("/{id}")
  public ResponseEntity<Book> getBookById(@PathVariable Long id) {
    Book book = bookRepository.findById(id).orElseThrow(() -> new RuntimeException("Book not found"));
    return ResponseEntity.ok(book);
  }

  @PutMapping("/{id}")
  public ResponseEntity<Book> updateBook(@PathVariable Long id, @RequestBody Book bookDetails) {
    Book book = bookRepository.findById(id).orElseThrow(() -> new RuntimeException("Book not found"));
    book.setTitle(bookDetails.getTitle());
    book.setAuthor(bookDetails.getAuthor());
    final Book updatedBook = bookRepository.save(book);
    return ResponseEntity.ok(updatedBook);
  }

  @DeleteMapping("/{id}")
  public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
    Book book = bookRepository.findById(id).orElseThrow(() -> new RuntimeException("Book not found"));
    bookRepository.delete(book);
    return ResponseEntity.noContent().build();
  }
}
```

6. Run Your Application

Start your application using your IDE or by running mvn spring-boot:run for Maven.

7. Access Swagger UI

Once your application is running, you can access the Swagger UI to interact with your API by visiting http://localhost:8080/swagger-ui.html in your web browser. This UI is automatically generated based on your OpenAPI configuration and will reflect the endpoints and models you've defined.

Using Redis

Integrating Redis into a Spring Boot application allows you to leverage its capabilities as a very fast, in-memory data store and cache. This can be particularly useful for enhancing the performance of RESTful services by caching common queries or storing session information. Below, I'll guide you through setting up Redis in a Spring Boot application step-by-step.

1. Set Up Your Spring Boot Project

Create a new Spring Boot project using Spring Initializr. Select your preferred dependencies including 'Spring Web' and 'Spring Data Redis'. You can also add 'Spring Boot DevTools' for automatic restarts and live reloads.

2. Add Redis Dependencies

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

3. Configure Redis in Spring Boot

application.properties:

```
spring.redis.host=localhost
spring.redis.port=6379
```

These settings assume you are running Redis locally on its default port (6379). If your Redis setup requires authentication or you are using a different port, you'll need to configure these properties accordingly.

4. Create a Redis Configuration Class

Create a configuration class to set up and customize the Redis template. This class will manage the data serialization and connection factory settings:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
public class RedisConfig {
```

```
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory connectionFactory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(connectionFactory);
        template.setKeySerializer(new StringRedisSerializer());
        template.setValueSerializer(new GenericJackson2JsonRedisSerializer());
        return template;
    }
}
```

This configuration sets up a RedisTemplate configured for JSON serialization of values.

5. Implement Caching or Direct Redis Access

You can directly use RedisTemplate to store and retrieve data. Here's an example of how you might use it within a Spring REST controller:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import org.springframework.data.redis.core.RedisTemplate;
import java.util.concurrent.TimeUnit;

@RestController
@RequestMapping("/api/cache")
public class CacheController {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    @GetMapping("/{key}")
    public Object getFromCache(@PathVariable String key) {
        return redisTemplate.opsForValue().get(key);
    }

    @PostMapping("/{key}")
    public String addToCache(@PathVariable String key, @RequestBody Object value) {
        redisTemplate.opsForValue().set(key, value, 10, TimeUnit.MINUTES);
        return "Value added to cache";
    }
}
```

In this controller, data can be stored in and retrieved from Redis, with the values expiring after 10 minutes.

6. Run and Test Your Application

Run your Spring Boot application using your IDE or by running mvn spring-boot:run for Maven. You can then test the REST endpoints using a tool like Postman.

Implementing JWT for Security

Incorporating JWT (JSON Web Tokens) for authentication and Redis for session storage in a Spring REST service is a robust solution for managing user sessions securely and efficiently. In this guide, we'll create a Spring Boot application that uses JWT for securing endpoints and Redis to store token-related data, enhancing the management of stateless authentication.

1. Set Up Your Spring Boot Application

Create a new Spring Boot project. You can use Spring Initializr (https://start.spring.io/) to generate your project. Choose dependencies for 'Spring Web', 'Spring Data Redis', 'Spring Security', and an appropriate JDBC dependency if your application requires a database.

2. Add Dependencies

Update your pom.xml to include dependencies for JWT and Redis:

```
<dependencies>
  <!-- Spring Boot Starter Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!-- Spring Boot Starter Security -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <!-- Spring Data Redis -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
  </dependency>
  <!-- JWT -->
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
  </dependency>
  <!-- Embedded Redis for local testing -->
  <dependency>
    <groupId>it.ozimov</groupId>
    <artifactId>embedded-redis</artifactId>
    <version>0.7.3</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

3. Configure Redis

application.properties

```
# Redis Configuration
spring.redis.host=localhost
spring.redis.port=6379
```

4. Create a Redis Configuration

Create a Redis configuration class for managing Redis operations:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
public class RedisConfig {

  @Bean
  public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory connectionFactory) {
    RedisTemplate<String, String> template = new RedisTemplate<>();
    template.setConnectionFactory(connectionFactory);
    template.setKeySerializer(new StringRedisSerializer());
    template.setValueSerializer(new StringRedisSerializer());
    return template;
  }
}
```

5. Configure Spring Security and JWT

Set up Spring Security with JWT:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
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;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
  @Autowired
  private UserDetailsService userDetailsService;

  @Bean
  public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
  }

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
      .authorizeRequests()
      .antMatchers("/api/public/**").permitAll()
      .anyRequest().authenticated()
      .and()
      .addFilter(new JWTAuthenticationFilter(authenticationManager()));
  }

  @Override
  @Bean
  public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
  }
}
```

Implement a filter for JWT authentication:

```
import com.fasterxml.jackson.databind.ObjectMapper;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import javax.servlet.FilterChain;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Date;

public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFilter {

  @Override
  public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
throws AuthenticationException {
```

```
    try {
      User creds = new ObjectMapper()
          .readValue(request.getInputStream(), User.class);
      return getAuthenticationManager().authenticate(
          new UsernamePasswordAuthenticationToken(
              creds.getUsername(),
              creds.getPassword(),
              new ArrayList<>())
      );
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }

  @Override
  protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
FilterChain chain,
                          Authentication auth) throws IOException, ServletException {
    String token = Jwts.builder()
        .setSubject(((User) auth.getPrincipal()).getUsername())
        .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
        .signWith(SignatureAlgorithm.HS512, SECRET)
        .compact();
    response.addHeader(HEADER_STRING, TOKEN_PREFIX + token);
  }
}
```

6. Implement UserDetailsService

Implement UserDetailsService to load user-specific data:

```
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {
  @Override
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    // Implement user lookup logic
    return new User("admin", "password", new ArrayList<>());
  }
}
```

7. Create REST Controller

Create a simple controller to handle login and test secured endpoints:

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

@RestController
public class UserController {

  @PostMapping("/login")
  public String login() {
    return "Login Successful";
  }

  @GetMapping("/secure")
  public String secureEndpoint() {
    return "Secure Endpoint";
  }
}
```

8. Run Your Application

Start your Spring Boot application. Test the /login endpoint using tools like Postman. After logging in, use the returned JWT in the Authorization header to access the /secure endpoint.

Front-end Frameworks using NodeJS platform

Introduction to NodeJS Platform

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside a web browser. It was created by Ryan Dahl in 2009, primarily to build network programs like web servers. The most significant feature of Node.js is that it enables developers to use JavaScript for server-side scripting—running scripts server-side to produce dynamic web page content before the page is sent to the user's web browser.

Core Features of Node.js

- Asynchronous and Event-Driven: All APIs of the Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js-based server never waits for an API to return data. The server moves to the next API after calling it, and a notification mechanism of Node.js helps the server to get a response from the previous API call.
- Single-Threaded but Highly Scalable: Node.js uses a single-threaded model with event looping. This event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests.
- No Buffering: Node.js applications never buffer any data. These applications simply output the data in chunks.

How Node.js Works?

Node.js uses the "event loop" as its runtime model. Everything in Node.js is basically handled by events. Node.js uses an event-driven model and callbacks extensively. It operates on a single thread, using non-blocking I/O calls, allowing it to handle tens of thousands of concurrent connections, which translates into high performance and scalability.

Why Use Node.js?

- Fast Execution: Node.js uses the V8 JavaScript Engine which makes it incredibly fast.
- Real-time Web Applications: Ideal for building real-time web applications like chat and gaming apps.
- Data Streaming: Good for applications that require real-time data streaming.
- Community: Large development community and a wealth of available modules and packages via the Node Package Manager (NPM).
- Unified Programming Language: Using JavaScript on both frontend and backend helps to perform more efficiently and reduces the context-switching time.

Getting Started with Node.js

Installation

You can download and install Node.js from its official website (nodejs.org). It is available for a variety of platforms like Windows, Linux, and macOS. Node.js packages come with NPM, which is used to install libraries and manage additional packages.

A Simple Example

Once Node.js is installed, you can write your first "Hello World" server with just a few lines of code:

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
```

```
  res.end('Hello World\n');
});

server.listen(3000, '127.0.0.1', () => {
  console.log('Server running at http://127.0.0.1:3000/');
});
```

Save this in a file named app.js, and run it by executing node app.js in your terminal. This script starts a server listening on port 3000, and it will respond with "Hello World" for every request.

The NodeJS Architecture

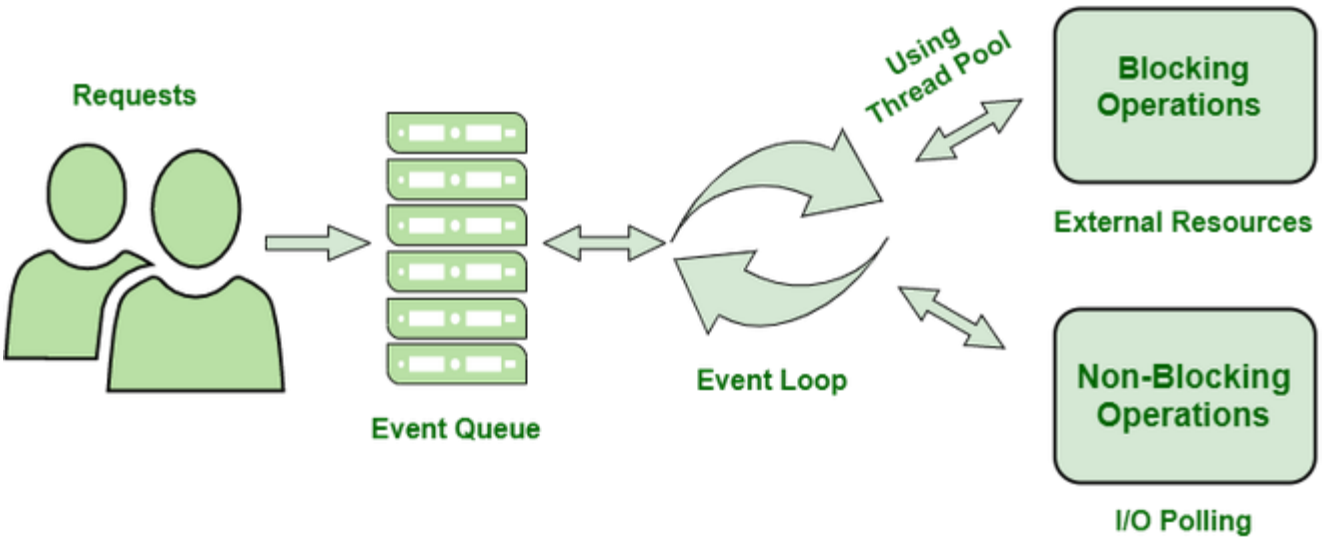
To manage several concurrent clients, Node.js employs a “Single Threaded Event Loop” design. The JavaScript event-based model and the JavaScript callback mechanism are employed in the Node.js Processing Model. It employs two fundamental concepts:

- Asynchronous model
- Non-blocking of I/O operations

These features enhance the scalability, performance, and throughput of Node.js web applications.

Components of the Node.js Architecture:

- ✓ Requests: Depending on the actions that a user needs to perform, the requests to the server can be either blocking (complex) or non-blocking (simple).
- ✓ Node.js Server: The Node.js server accepts user requests, processes them, and returns results to the users.
- ✓ Event Queue: The main use of Event Queue is to store the incoming client requests and pass them sequentially to the Event Loop.
- ✓ Thread Pool: The Thread pool in a Node.js server contains the threads that are available for performing operations required to process requests.
- ✓ Event Loop: Event Loop receives requests from the Event Queue and sends out the responses to the clients.
- ✓ External Resources: In order to handle blocking client requests, external resources are used. They can be of any type (computation, storage, etc).



Installation and configuration

Windows:

1. Download the Installer:
    a. Go to the official Node.js website (nodejs.org).
    b. Click on the "Windows Installer" link to download the .msi installer (choose either the LTS or Current version depending on your needs for stability or latest features).
2. Run the Installer:
    a. Execute the downloaded .msi file.
    b. Follow the prompts in the Setup Wizard. Make sure to agree to the license agreement and select the installation path.
    c. Select the components to install. Typically, you should install the Node.js runtime, npm package manager, and optionally, the necessary tools and binaries to build native modules.
3. Finish the Installation:
    a. Click the Finish button to complete the installation.
4. Verify Installation:
    a. Open a command prompt or PowerShell, and type:
    b. This command will print the Node.js version, confirming its installation.

```
node -v
```

    c. This command shows the installed version of npm, the Node package manager.

```
npm -v
```

Configuration

After installation, you might want to configure npm's global packages' installation directory or handle proxy settings.

Set Global Installation Directory:

To change the directory where npm installs global packages, you can configure npm's prefix setting:

```
npm config set prefix ~/npm
```

Add this new directory to your PATH so that globally installed packages will be executable:

```
export PATH="$PATH:$HOME/npm/bin"
```

Add this line to your .bashrc, .bash_profile, or .zshrc file to make the change permanent.

Proxy Configuration:

If you are behind a proxy, configure npm to work with it:

```
npm config set proxy http://proxy-server-url:port
npm config set https-proxy http://proxy-server-url:port
```

The node CLI commands

1. Checking Node.js and npm Versions
Before starting development, it's often necessary to check the installed versions of Node.js and npm (Node package manager).

Node.js Version:

```
node -v
```

or

```
node --version
```

npm Version:

```
npm -v
```

or

```
npm --version
```

2. Running a Node.js Script
To run a JavaScript file with Node.js, use the node command followed by the file name:

```
node app.js
```

This will execute the code in app.js using the Node.js runtime.

3. Interactive REPL
Node.js provides an interactive Read-Eval-Print Loop (REPL) that can be used for experimenting with Node.js commands and for quick computations.

Start the REPL:

```
node
```

Once in the REPL, you can type JavaScript code directly into your terminal, and the results will be output immediately.

4. Initializing a New Node.js Project
To start a new Node.js project, you can initialize a new package.json file, which will hold metadata about your project as well as list its dependencies.

Initialize Project:

```
npm init
```

This command will prompt you to enter several pieces of information (like project name, version, description), or you can use:

```
npm init -y
```

to generate a package.json file with default values without being prompted for input.

5. Installing Packages
npm is used to manage libraries and tools for your Node.js application.

Install a Package Locally:

```
npm install <package_name>
```

This will install the package and save it in the node_modules directory as well as list it as a dependency in your package.json.

Install a Package Globally:

```
npm install -g <package_name>
```

Installing a package globally allows you to use it from anywhere on your system.

6. Updating and Uninstalling Packages

Update a Package:

```
npm update <package_name>
```

This will update the specified package within the version limits set in your package.json.

Uninstall a Package:

```
npm uninstall <package_name>
```

This removes the package from your node_modules directory and updates your package.json and package-lock.json.

7. Listing Installed Packages

List All Installed Packages:

```
npm list
```

This command will show you a tree of all installed packages.

List Top-Level Installed Packages:

```
npm list --depth=0
```

This shows only the packages that you have explicitly installed (i.e., not their dependencies).

8. Managing Node.js Versions
If you work with multiple versions of Node.js, you might need a version manager like nvm (Node Version Manager).

Install nvm: (See the nvm GitHub page for installation instructions.)
Install a Specific Node.js Version with nvm:

```
nvm install <version>
```

Switch Between Installed Node.js Versions:

```
nvm use <version>
```

List Installed Node.js Versions:

```
nvm ls
```

Creating the NodeJS project

To start a new Node.js project, follow these steps:

1. Install Node.js

Download and install Node.js from nodejs.org. This will also install npm, which is Node.js's package manager.

2. Create a Project Directory

Create a new directory for your project and navigate into it:

```
mkdir my-node-project
cd my-node-project
```

3. Initialize a new Node.js project

Run

```
npm init
```

to create a package.json file.

Follow the prompts to specify the details of your project.

You can use

```
npm init -y
```

to generate it with default values without going through an interactive setup.

4. Create Your First Script

Create a file named app.js (or another name if you prefer) and write a simple JavaScript code:

javascript

```
console.log('Hello, Node.js!');
```

Run your script from the command line:

```
node app.js
```

You should see "Hello, Node.js!" printed to the terminal.

The directory structure

A typical Node.js project might have the following directory structure:

```
my-node-project/
|
├── node_modules/        # Directory where npm packages are stored
|
├── src/                 # Source files for your project
|   └── app.js           # Main application script
|
├── test/                # Test scripts
|
├── package.json         # Project manifest file
|
├── package-lock.json    # Automatically generated for any operations
|
└── README.md            # Project overview
```

The package.json file

The package.json file is the heart of a Node.js project. It holds metadata relevant to the project and it is used to manage the project's dependencies, scripts, versioning, and a whole lot more.

Here is an example package.json:

```
{
  "name": "my-node-project",
  "version": "1.0.0",
  "description": "A sample Node.js project",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "author": "Your Name",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

The project versioning specification

Node.js projects typically follow Semantic Versioning, or SemVer, which is structured as MAJOR.MINOR.PATCH:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards compatible manner, and
- PATCH version when you make backwards compatible bug fixes.

The node global variables

Node.js provides several global variables that are available in all modules. Some of the most commonly used globals are:

- __dirname: The directory name of the current module (equals path.dirname(__filename)).
- __filename: The file name of the current module.
- require: Function to use modules (CommonJS).
- module: A reference to the current module.
- process: Provides information about, and control over, the current Node.js process.

The node errors

Handling errors in Node.js is crucial for building reliable applications. Node.js errors can be handled using try...catch statements or by using asynchronous error handling mechanisms like callbacks and promises.

Example of try...catch:
javascript

```
try {
  // Code that may throw an error
  const m = 1;
  const n = m + z; // ReferenceError: z is not defined
} catch (err) {
  // Code to execute after catching an error
  console.error(` Error: ${err}` );
}
```

Example of Asynchronous Error Handling with Promises:
javascript

```
function getSomething() {
  return new Promise((resolve, reject) => {
    // Something went wrong
    reject(new Error("Failed to get something"));
  });
}

getSomething().then(data => {
  console.log(data);
}).catch(err => {
  console.error(err.message);
});
```

Learning the NodeJS built-in modules

The path module

The path module provides utilities for working with file and directory paths. It helps manage path strings in a consistent way across different operating systems.

Example:

```
const path = require('path');

// Normalize a path
const normalizedPath = path.normalize('/users/john/docs//new/../doc.txt');
console.log(normalizedPath);

// Join several paths
const joinedPath = path.join('/users', 'john', 'docs', 'doc.txt');
console.log(joinedPath);

// Resolve a path
const resolvedPath = path.resolve('app.js');
console.log(resolvedPath);

// Get the directory name of a path
const dirName = path.dirname('/users/john/docs/doc.txt');
console.log(dirName);

// Get the base name of a path
const baseName = path.basename('/users/john/docs/doc.txt');
console.log(baseName);

// Get the extension of a path
const extName = path.extname('/users/john/docs/doc.txt');
console.log(extName);
```

more sample usage

```
const path = require('path');

// Example 1: Normalize a path
console.log('Normalize:', path.normalize('/users/john/..//test.txt'));

// Example 2: Join paths together
console.log('Join:', path.join('/foo', 'bar', 'baz/asdf', 'quux', '..'));

// Example 3: Resolve an absolute path
console.log('Resolve:', path.resolve('test.txt'));

// Example 4: Extract the extension of a file
console.log('Extension:', path.extname('index.html'));
```

```
// Example 5: Get the directory name from a path
console.log('Dirname:', path.dirname('/foo/bar/baz/asdf/quux'));
```

The os module

The os module provides utilities related to the operating system. It's useful for retrieving information about the runtime environment.

Example:

```
const os = require('os');

// Get the operating system platform
console.log('OS Platform:', os.platform());

// Get the total memory
console.log('Total Memory:', os.totalmem());

// Get the free memory
console.log('Free Memory:', os.freemem());

// Get the number of CPU cores
console.log('CPUs:', os.cpus().length);

// Get the system uptime
console.log('Uptime:', os.uptime());

// Get the network interfaces
console.log('Network Interfaces:', os.networkInterfaces());

// Get the OS release
console.log('OS Release:', os.release());
```

The http module

The http module allows you to create HTTP servers and clients. It's one of the foundational modules for web services in Node.js.

Example:

```
const http = require('http');

// Create an HTTP server
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
```

```
});
```

More sample usage:

```
const http = require('http');

// Example 1: Create HTTP server and respond with 'Hello World'
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});
server.listen(3000, () => console.log('Server running on http://localhost:3000/'));

// Example 2: Make an HTTP GET request
http.get('http://api.example.com/data', (resp) => {
  let data = '';
  resp.on('data', (chunk) => data += chunk);
  resp.on('end', () => console.log(data));
});

// Example 3: HTTP Server responding with JSON
const jsonServer = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({ message: 'Hello JSON' }));
});
jsonServer.listen(3001);

// Example 4: POST Data to an HTTP server
const postData = JSON.stringify({ 'msg': 'Hello World!' });
const postOptions = {
  hostname: 'www.example.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
     'Content-Type': 'application/json',
     'Content-Length': Buffer.byteLength(postData)
  }
};
const postReq = http.request(postOptions, (res) => {
  res.setEncoding('utf8');
  res.on('data', (chunk) => console.log(` Response: ${chunk}` ));
});
postReq.write(postData);
postReq.end();

// Example 5: Handling HTTP request errors
const req = http.get('http://example.com', (response) => {
  // handle the response
}).on('error', (e) => {
```

```
  console.error(`Got error: ${e.message}`);
});
```

The commonjs module

CommonJS is a standard for modularizing JavaScript that Node.js uses. A module encapsulated in CommonJS can easily export and import dependencies.

Example:
Create two files, greet.js and app.js.

greet.js:

```
function sayHello(name) {
  return `Hello, ${name}!`;
}

module.exports = sayHello;
```

app.js:

```
const greet = require('./greet');
console.log(greet('Alice'));
```

Run node app.js to see the output.

More sample usage:

```
// Example 1: Exporting a single function in a module
// In greet.js
module.exports = function greet(name) {
   return `Hello, ${name}!`;
};

// In app.js
const greet = require('./greet');
console.log(greet('Alice'));

// Example 2: Exporting an object containing multiple functions
// In math.js
module.exports = {
   add: (a, b) => a + b,
   subtract: (a, b) => a - b
};

// In app.js
const math = require('./math');
console.log(math.add(5, 3));
console.log(math.subtract(5, 3));

// Example 3: Caching with modules (Node.js caches the first required module)
// In logger.js
```

```
console.log('Loading the module');
module.exports = function log(message) {
  console.log(message);
};

// In app.js
const logger = require('./logger');
logger('This is a message');

// Example 4: Overriding exports with a new object
// In user.js
module.exports = {
  name: 'John',
  age: 30
};

// In app.js
const user = require('./user');
console.log(user.name);

// Example 5: Mutating the exports object
// In counter.js
let count = 0;
module.exports.increment = () => ++count;
module.exports.getCount = () => count;

// In app.js
const counter = require('./counter');
counter.increment();
console.log(counter.getCount()); // 1
```

The es module

ES Modules are the standard module system in JavaScript that allows you to use import and export.

Example:
Ensure your package.json includes "type": "module" to enable ES module support.

Create two files, greet.mjs and app.mjs.

greet.mjs:

```
export function sayHello(name) {
 return `Hello, ${name}!`;
}
```

app.mjs:

```
import { sayHello } from './greet.mjs';
console.log(sayHello('Alice'));
```

Run node app.mjs to see the output.

More sample usage:

```
// Example 1: Exporting and importing a single function
// In greet.js
export function greet(name) {
  return `Hello, ${name}!`;
}

// In app.js
import { greet } from './greet.js';
console.log(greet('Alice'));

// Example 2: Exporting multiple functions
// In math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// In app.js
import * as math from './math.js';
console.log(math.add(5, 3));
console.log(math.subtract(5, 3));

// Example 3: Exporting and importing a default function
// In greet.js
export default function(name) {
  return `Hello, ${name}!`;
}

// In app.js
import greet from './greet.js';
console.log(greet('Alice'));

// Example 4: Renaming imports
// In math.js
export const add = (a, b) => a + b;

// In app.js
import { add as addNumbers } from './math.js';
console.log(addNumbers(5, 3));

// Example 5: Dynamic imports
// In app.js
const moduleName = './greet.js';
import(moduleName)
 .then((module) => {
  console.log(module.greet('Alice'));
 });
```

The events module

The events module allows you to work with event-driven programming. It's used to fire and listen for your own events.

Example:

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});

myEmitter.emit('event');
```

More sample usage

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

// Example 1: Simple event listener
emitter.on('event', function() {
   console.log('An event occurred!');
});
emitter.emit('event');

// Example 2: Event with parameters
emitter.on('status', (code, msg) => console.log(` Got ${code} and ${msg}` ));
emitter.emit('status', 200, 'ok');

// Example 3: Multiple listeners
emitter.on('multi', () => console.log('First listener'));
emitter.on('multi', () => console.log('Second listener'));
emitter.emit('multi');

// Example 4: Only once listener
emitter.once('once', () => console.log('This will only fire once'));
emitter.emit('once');
emitter.emit('once');  // Will not execute

// Example 5: Removing listeners
const callback = () => console.log('I will be removed');
emitter.on('remove', callback);
emitter.removeListener('remove', callback);
emitter.emit('remove'); // No output
```

The fs module

The fs module is used for interacting with the file system.

Example:

```
const fs = require('fs');

// Write to a file asynchronously
fs.writeFile('message.txt', 'Hello Node.js', (err) => {
  if (err) throw err;
  console.log('The file has been saved!');

  // Read the file
  fs.readFile('message.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
  });
});
```

More sample usage:

```
const fs = require('fs');

// Example 1: Read file asynchronously
fs.readFile('file.txt', 'utf8', (err, data) => {
   if (err) throw err;
   console.log(data);
});

// Example 2: Write file asynchronously
fs.writeFile('file.txt', 'Hello Node!', (err) => {
   if (err) throw err;
   console.log('File written!');
});

// Example 3: Append to file asynchronously
fs.appendFile('file.txt', ' More text', (err) => {
   if (err) throw err;
   console.log('Updated!');
});

// Example 4: Delete file asynchronously
fs.unlink('file.txt', (err) => {
   if (err) throw err;
   console.log('File deleted!');
});

// Example 5: Rename file asynchronously
fs.rename('file.txt', 'newfile.txt', (err) => {
   if (err) throw err;
   console.log('File renamed!');
```

```
});
```

Managing custom modules

In Node.js, managing custom modules is a straightforward and essential skill for structuring larger applications and sharing code between files or across different projects. A custom module can be anything from a simple utility library to a large set of related functions and objects. Here's how you can create, export, import, and manage these custom modules.

1. Create a Custom Module

To start, create a new file that will contain your module. This file will hold the logic that you want to export and use in other parts of your application. For example, let's create a simple module for basic math operations.

Create a file named mathUtils.js:

```
// Define some math functions
function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return a - b;
}

function multiply(a, b) {
  return a * b;
}

function divide(a, b) {
  if (b === 0) {
    throw new Error("Cannot divide by zero.");
  }
  return a / b;
}

// Export the functions
module.exports = {
  add,
  subtract,
  multiply,
  divide
};
```

2. Importing the Custom Module

To use the functions defined in your module, you will need to import the module into the file where you want to use it. Use the require() function to do this.

Create a file named app.js:

```
const mathUtils = require('./mathUtils');

const sum = mathUtils.add(10, 5);
const difference = mathUtils.subtract(10, 5);
const product = mathUtils.multiply(10, 5);
const quotient = mathUtils.divide(10, 5);

console.log(`Sum: ${sum}`);
console.log(`Difference: ${difference}`);
console.log(`Product: ${product}`);
console.log(`Quotient: ${quotient}`);
```

3. Running Your Application

To run your application, use the Node.js runtime:

```
node app.js
```

This should output the results of the math operations to the console.

4. Managing Module Logic

As your application grows, you may find that your modules become more complex. It's a good practice to keep them organized and focused on a single responsibility. For instance, if your mathUtils module gets too large, consider splitting it into smaller modules like arithmeticOperations.js, trigonometricFunctions.js, etc.

5. Publishing a Module

If you create a module that you think could be useful to others, you can publish it to npm, the Node package manager. Here's a quick rundown on how to do it:

Create a package.json file in your module's root directory if you haven't already. You can do this manually or by running npm init and following the prompts.

Login to your npm account from the command line (you can create one at npmjs.com if you don't have one):

```
npm login
```

Publish your module:

```
npm publish
```

Make sure your package name is unique, or npm will reject your publish request.

6. Versioning Your Module

If you publish a module and later make updates to it, you'll need to manage versions carefully. Follow semantic versioning (semver) principles:

- Increment the patch version for backwards-compatible bug fixes.
- Increment the minor version for new functionality that is backwards-compatible.
- Increment the major version for changes that make your new version incompatible with previous versions.
- Update the version in your package.json file, and then run npm publish again to update the module on npm.

Introduction to express.js framework

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node-based web applications. Below is a detailed guide on getting started with Express.js, including installation, basic setup, routing, middleware, and a simple REST API example.

1. Install Node.js

Before you can start using Express.js, you need to have Node.js installed on your computer. If you haven't installed Node.js yet, download and install it from nodejs.org.

2. Setup a New Node.js Project

Once Node.js is installed, you can set up a new Node.js project.

Create a new directory for your project and navigate into it:

```
mkdir my-express-app
cd my-express-app
```

Initialize a new Node.js project:

```
npm init -y
```

This command creates a package.json file with default values.

3. Install Express.js

Now, install Express.js within your project directory by running:

```
npm install express
```

This command adds Express as a dependency in your project's package.json file and downloads the Express library to the node_modules directory.

4. Create Your Main Server File

Create a file named app.js in your project directory. This file will contain your Express application.

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(` Example app listening at http://localhost:${port}` );
});
```

In this simple server setup:

- require('express') imports the Express module.
- express() creates an Express application.
- .get('/') defines a route handler for GET requests to the root URL (/).
- res.send('Hello World!') sends back a response to the client.
- app.listen() starts a server and listens on the specified port for connections.

5. Run Your Express Server

Run your application using Node.js:

```
node app.js
```

Now, if you open a web browser and go to http://localhost:3000, you should see the message "Hello World!"

6. Basic Routing

Express provides methods to specify what function is called for a particular HTTP verb (GET, POST, etc.) and URL pattern ("route"). Here is an example showing basic routing:

```
app.get('/about', (req, res) => {
  res.send('About Page');
});

app.post('/submit', (req, res) => {
  res.send('Form Submitted');
});
```

7. Using Middleware

Middleware functions can execute any code, make changes to the request and the response objects, end the request-response cycle, and call the next middleware function.

Here's an example of a simple middleware that logs the request URL:

```
app.use((req, res, next) => {
  console.log('Request URL:', req.originalUrl);
  next();
});
```

8. Setting Up a Simple REST API

Express simplifies the process of building server-side routes to handle client requests. Here's how you can set up a basic REST API:

```
app.get('/api/users', (req, res) => {
  res.status(200).json({ users: ['User1', 'User2'] });
});

app.get('/api/users/:id', (req, res) => {
  res.status(200).json({ id: req.params.id, name: "User1" });
});
```

9. Testing Your API

You can test your Express API routes using tools like Postman or curl to make requests and see the responses.

Implementing REST APIs with express.js

Demo: Creating the Express.js REST API

1. Initialize the Project

Create a new directory for your Express.js project and initialize it with npm:

```
mkdir express-api
cd express-api
npm init -y
```

2. Install Necessary Packages

Install Express and other useful packages like body-parser (to parse incoming request bodies):

```
npm install express body-parser
```

3. Set Up the Express Server

Create a file named app.js:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
```

```
const port = 3000;

app.use(bodyParser.json());

// In-memory database
const books = [];

// POST: Create a book
app.post('/books', (req, res) => {
  const book = req.body;
  book.id = books.length + 1;
  books.push(book);
  res.status(201).send(book);
});

// GET: Read all books
app.get('/books', (req, res) => {
  res.status(200).send(books);
});

// GET: Read a book by ID
app.get('/books/:id', (req, res) => {
  const book = books.find(b => b.id === parseInt(req.params.id));
  if (!book) res.status(404).send('Book not found');
  else res.send(book);
});

// PUT: Update a book
app.put('/books/:id', (req, res) => {
  const book = books.find(b => b.id === parseInt(req.params.id));
  if (!book) res.status(404).send('Book not found');
  else {
    book.title = req.body.title;
    book.author = req.body.author;
    res.send(book);
  }
});

// DELETE: Delete a book
app.delete('/books/:id', (req, res) => {
  const index = books.findIndex(b => b.id === parseInt(req.params.id));
  if (index >= 0) {
    books.splice(index, 1);
    res.send({ message: 'Book deleted' });
  } else {
    res.status(404).send('Book not found');
  }
});

app.listen(port, () => {
```

```
  console.log(`Server running on http://localhost:${port}`);
});
```

This server includes endpoints for creating, retrieving, updating, and deleting books. It uses an array to store book data in memory.

4. Run Your Express Server

Run your server with the following command:

```
node app.js
```

Consuming REST services from Java web services

Part 1: Creating the Spring Boot REST API

1. Set Up Spring Boot Project

Use Spring Initializr (https://start.spring.io/) to generate a new Spring Boot project. Select Maven or Gradle as the build tool and add dependencies for 'Spring Web', 'Spring Data JPA', and 'H2 Database' for in-memory data storage.

2. Define a Simple Entity and Repository

Create a Book entity and a corresponding repository.

Model (Book.java):

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

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 constructors, getters, and setters
}
```

Repository (BookRepository.java):

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

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

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

3. Create a REST Controller for CRUD Operations

Implement a REST controller that provides endpoints for managing books.

Controller (BookController.java):

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

import com.example.demo.model.Book;
import com.example.demo.repository.BookRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

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

  @Autowired
  private BookRepository bookRepository;

  @GetMapping
  public List<Book> getAllBooks() {
    return bookRepository.findAll();
  }

  @GetMapping("/{id}")
  public Book getBookById(@PathVariable Long id) {
    return bookRepository.findById(id).orElse(null);
  }

  @PostMapping
  public Book createBook(@RequestBody Book book) {
    return bookRepository.save(book);
  }

  @PutMapping("/{id}")
  public Book updateBook(@PathVariable Long id, @RequestBody Book bookDetails) {
    return bookRepository.findById(id)
      .map(book -> {
```

```
            book.setTitle(bookDetails.getTitle());
            book.setAuthor(bookDetails.getAuthor());
            return bookRepository.save(book);
        }).orElseGet(() -> {
            bookDetails.setId(id);
            return bookRepository.save(bookDetails);
        });
    }

    @DeleteMapping("/{id}")
    public void deleteBook(@PathVariable Long id) {
        bookRepository.deleteById(id);
    }
}
```

4. Run Your Spring Boot Application

Run the application, and your API will be available at http://localhost:8080/api/books.

Part 2: Create the Express.js Application to Consume the Spring Boot API

1. Set Up the Express.js Project

Initialize your Express.js project as previously described and install the required packages:

```
npm init -y
npm install express axios
```

2. Implement CRUD Operations in Express.js to Consume the API

Create app.js in the Express.js project:

```
const express = require('express');
const axios = require('axios');
const app = express();
const port = 3001;  // Make sure this is different from the Spring Boot port

app.use(express.json());

const apiUrl = 'http://localhost:8080/api/books';

app.get('/books', async (req, res) => {
    try {
        const response = await axios.get(apiUrl);
        res.json(response.data);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
});

app.get('/books/:id', async (req, res) => {
    try {
```

```
    const response = await axios.get(`${apiUrl}/${req.params.id}`);
    res.json(response.data);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

app.post('/books', async (req, res) => {
  try {
    const response = await axios.post(apiUrl, req.body);
    res.json(response.data);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

app.put('/books/:id', async (req, res) => {
  try {
    const response = await axios.put(`${apiUrl}/${req.params.id}`, req.body);
    res.json(response.data);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

app.delete('/books/:id', async (req, res) => {
  try {
    const response = await axios.delete(`${apiUrl}/${req.params.id}`);
    res.json({ message: 'Book deleted' });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

app.listen(port, () => {
  console.log(`Express server running at http://localhost:${port}`);
});
```

3. Run Your Express.js Application

Start the Express.js server:

```
node app.js
```

Testing

You can now test the Express.js endpoints, which internally call the Spring Boot service. Use tools like Postman or cURL to make requests to the Express.js server at http://localhost:3001/books and observe how it interacts with the Spring Boot backend.

Introduction to FeatherJS framework

FeathersJS is a web framework for building real-time applications and REST APIs using JavaScript or TypeScript. It wraps Express.js, allowing you to use most Express middleware, but adds additional features to support real-time capabilities and simplifies data handling with flexible plugins. It's designed to be a minimalist framework that can control the flow of data through RESTful resources and real-time applications using WebSockets.

How to Install and Configure FeathersJS

1. Installing FeathersJS

To get started with FeathersJS, you first need to have Node.js installed on your system. If Node.js is already installed, you can create a new FeathersJS application using the Feathers CLI. First, install the CLI:

```
npm install -g @feathersjs/cli
```

2. Create a New FeathersJS Application

Once the CLI is installed, you can create a new application by running:

```
feathers generate app
```

Follow the prompts to set up your project. For instance, you can choose to use JavaScript or TypeScript, select which testing framework you'd like to use (like Mocha), and configure other settings such as the linter.

3. Running the FeathersJS Application

Navigate into your new project directory and start the application:

```
cd path-to-your-new-app
npm start
```

By default, your new FeathersJS application will run on http://localhost:3030.

Implementing express.js API using FeatherJS

Demo: Implementing the Express.js API Using FeathersJS

We'll convert the Express.js example from our previous activity, which consumed a Java Spring Boot REST service, into a FeathersJS application.

1. Set Up the FeathersJS Application

Assuming you have generated your FeathersJS application as described above, you will now need to add Axios for making HTTP requests to the Spring Boot service:

```
npm install axios
```

2. Create a Custom Service in FeathersJS

Instead of using the CLI to generate a service, we'll manually create one to consume the Spring Boot API.

Create a new folder and file for our service:

```
mkdir src/services/book
touch src/services/book/book.service.js
```

Now, define the service in book.service.js:

```
const axios = require('axios');
const apiUrl = 'http://localhost:8080/api/books';

class BookService {
 async find(params) {
  const response = await axios.get(apiUrl);
  return response.data;
 }

 async get(id, params) {
  const response = await axios.get(`${apiUrl}/${id}`);
  return response.data;
 }

 async create(data, params) {
  const response = await axios.post(apiUrl, data);
  return response.data;
 }

 async update(id, data, params) {
  const response = await axios.put(`${apiUrl}/${id}`, data);
  return response.data;
 }

 async patch(id, data, params) {
  return this.update(id, data, params);
 }

 async remove(id, params) {
  const response = await axios.delete(`${apiUrl}/${id}`);
  return response.data;
 }
}

module.exports = function (app) {
 app.use('/books', new BookService());
};
```

3. Register the Service

You need to register this new service in your FeathersJS application. Edit src/services/index.js and add:

```
const books = require('./book/book.service.js');

module.exports = function (app) {
  app.configure(books);
};
```

4. Start and Test Your FeathersJS Application

Run your FeathersJS application:

```
npm start
```

Now, you can access your FeathersJS application's endpoints, e.g., http://localhost:3030/books, which will internally call the Spring Boot service endpoints.

Introduction to Vue 3/4 framework

Vue.js is a progressive JavaScript framework used for building user interfaces. It is designed to be incrementally adoptable, meaning that it can be easily integrated into projects where only parts of the application need Vue capabilities.

Vue is primarily focused on the view layer only, making it easy to pick up and integrate with other libraries or existing projects. Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries.

Installation and Configuration

Using a CDN:
For quick prototyping, you can include Vue via a CDN:

```
<script src="https://cdn.jsdelivr.net/npm/vue@3"></script>
```

NPM:
For real projects, it's best to install Vue via npm for better version control and module bundling:

```
npm install vue@next
```

CLI Installation:
Install Vue CLI globally for scaffolding new projects:

```
npm install -g @vue/cli
```

Create a Project:

Create and start a new project using Vue CLI:

```
vue create my-vue-app
cd my-vue-app
npm run serve
```

Vite:
You can also set up a Vue project using Vite, which is faster for development:

```
npm init vite@latest my-vue-app -- --template vue
cd my-vue-app
npm install
npm run dev
```

Vue.js Components

Components are reusable Vue instances with a name:

Global Registration:

```
// Define a new global component called 'my-component'
Vue.component('my-component', {
  template: `<div>A custom component!</div>`
});
```

Local Registration:

```
// Define a local component
const MyComponent = {
  template: `<div>Another custom component!</div>`
};
new Vue({
  el: '#app',
  components: {
    'my-component': MyComponent
  }
});
```

Single File Components:

```
In MyComponent.vue:
<template>
  <div>{{ message }}</div>
</template>
<script>
export default {
  data() {
    return {
      message: 'Hello from Component'
    };
  }
```

```
}
</script>
```

Props:

```
<template>
  <div>{{ title }}</div>
</template>
<script>
export default {
  props: ['title']
}
</script>
```

Events:

```
<template>
  <button @click="onClick">Click me!</button>
</template>
<script>
export default {
  methods: {
    onClick() {
      this.$emit('custom-event', 'data');
    }
  }
}
</script>
```

Watch Properties

Watchers are used to perform actions when a data property changes:

Basic Watch:

```
new Vue({
  data() {
    return { counter: 0 };
  },
  watch: {
    counter(newVal, oldVal) {
      console.log(` Counter changed from ${oldVal} to ${newVal}` );
    }
  }
});
```

Creating Templates

Templates define the markup of Vue components:

Using Template Syntax:

```
<template>
  <div>{{ message }}</div>
</template>
<script>
export default {
  data() {
    return {
      message: 'Hello Vue!'
    };
  }
}
</script>
```

Binding Data

Data binding is a way to bind data to DOM elements:

Text Binding:

```
<template>
  <div>{{ message }}</div>
</template>
```

Attribute Binding:

```
<template>
  <div :id="dynamicId">Hello!</div>
</template>
```

Rendering Data

Vue can render lists and conditionals within templates:

Conditional Rendering:

```
<template>
  <div v-if="seen">Now you see me</div>
</template>
```

List Rendering:

```
<template>
  <ul>
    <li v-for="item in items" :key="item.id">{{ item.text }}</li>
  </ul>
</template>
```

Applying Events

Handling user interactions with event handlers:

Listening to Events:

```
<template>
 <button @click="greet">Greet</button>
</template>
<script>
export default {
 methods: {
  greet() {
   alert('Hello!');
  }
 }
}
</script>
```

Consuming REST services using Vuex, Axios and Pinia

Demo: Creating a Vue 3 application that consumes a CRUD REST service from FeathersJS using Vuex, Axios, and Pinia

Setting Up the Vue 3 Project

1. Install Vue CLI
   First, ensure that you have Vue CLI installed:

```
npm install -g @vue/cli
```

2. Create a New Vue Project
   Generate a new Vue 3 project:

```
vue create vue-feathers-client
```

Choose "Manually select features", make sure to select:

- ✓ Babel
- ✓ Vuex
- ✓ Router
- ✓ Pinia
- ✓ Linter / Formatter

Configure each tool according to your preference when prompted.

3. Install Axios and Set Up Environment

   Navigate into your new project directory and install Axios:

```
cd vue-feathers-client
npm install axios
```

4. Set Up Vuex, Pinia, and Axios

Create a src/api/axios.js to configure Axios specifically for interaction with the FeathersJS backend:

```
import axios from 'axios';

const apiClient = axios.create({
 baseURL: 'http://localhost:3030', // Adjust this URL to the location of your FeathersJS backend
 withCredentials: false, // This depends on your CORS setup
 headers: {
  Accept: 'application/json',
  'Content-Type': 'application/json'
 }
});

export default apiClient;
```

Configure Pinia Store for CRUD Operations

1. Setup Pinia Store

   Setup Pinia store src/stores/books.js:

```
import { defineStore } from 'pinia';
import axios from '../api/axios';

export const useBookStore = defineStore('books', {
 state: () => ({
  books: []
 }),
 actions: {
  fetchBooks() {
   return axios.get('/books')
    .then(response => {
     this.books = response.data;
    })
    .catch(error => {
     throw new Error('Error fetching books: ' + error);
    });
  },
  addBook(book) {
   return axios.post('/books', book)
    .then(response => {
     this.books.push(response.data);
    });
  },
```

```
  updateBook(id, book) {
   return axios.put(`/books/${id}`, book)
    .then(() => {
     const index = this.books.findIndex(b => b.id === id);
     if (index !== -1) {
      this.books.splice(index, 1, { ...book, id });
     }
    });
  },
  deleteBook(id) {
   return axios.delete(`/books/${id}`)
    .then(() => {
     const index = this.books.findIndex(b => b.id === id);
     if (index !== -1) {
      this.books.splice(index, 1);
     }
    });
  }
 }
});
```

Building the Vue Components

1. Create the BookList Component

Create a component src/components/BookList.vue:

```
<template>
 <div>
  <h1>Books</h1>
  <ul>
   <li v-for="book in books" :key="book.id">
    {{ book.title }} by {{ book.author }}
    <button @click="deleteBook(book.id)">Delete</button>
   </li>
  </ul>
  <button @click="fetchBooks">Refresh Books</button>
 </div>
</template>

<script>
import { useBookStore } from '@/stores/books';

export default {
 name: 'BookList',
 setup() {
  const bookStore = useBookStore();
  const books = bookStore.books;
```

```
  function fetchBooks() {
    bookStore.fetchBooks();
  }

  function deleteBook(id) {
    bookStore.deleteBook(id);
  }

  return { books, fetchBooks, deleteBook };
 }
}
</script>
```

2. Hook Up the Store in the Main App

Edit src/App.vue to use the BookList component:

```
<template>
 <div id="app">
  <BookList />
 </div>
</template>

<script>
import BookList from './components/BookList.vue';

export default {
 name: 'App',
 components: {
  BookList
 }
}
</script>
```

Running the Application

Run the Vue application:

```
npm run serve
```

This command compiles and hot-reloads for development. Make sure your FeathersJS server is running before you start your Vue application.

Note: This setup shows a complete Vue 3 application using Pinia for state management, Axios for API requests, consuming a FeathersJS REST API. This example provides CRUD functionality, enabling operations on a 'books' resource. Adjust the base URL in the Axios setup to match your FeathersJS API endpoint, and you should be able to perform full CRUD operations through your Vue.js frontend.