Implementing a Full REST Service

1. Setting the scene
2. Defining a full REST service

Demo project: **`demo-full-rest-services`**

1. Setting the Scene

- Overview
- Example REST controller
- Using Swagger to expose the REST API
- Using the Swagger UI

Demo package: `demo.restservices`

Overview

- So far, we've seen how to GET data from a REST service:

```
@GetMapping(...)
```

- Here's how to support the other HTTP verbs:

```
@PostMapping(...)
```

```
@PutMapping(...)
```

```
@DeleteMapping(...)
```

Example REST Controller

- Here's the example REST controller for our example:

```
@RestController
@RequestMapping("/full")
@CrossOrigin
public class FullController {

    @Autowired
    private ProductRepository repository;

    // Full CRUD API, see following slides
    …
}
```

FullController.java

- Note:
  - We've defined a repository bean to manage data persistence
  - See `ProductRepository.java` for details

Using Swagger to Expose the REST API (1 of 2)

- We'll use Swagger to help us test our REST API
  - Swagger is an open-source project
  - Enables you to document your REST API

- What does Swagger do?
  - Exposes metadata about your REST controller classes and paths
  - Enables you to test GET, POST, PUT, DELETE endpoints

- For full details about using Swagger in Spring Boot, see:
  - https://springdoc.org/

Using Swagger to Expose the REST API (2 of 2)

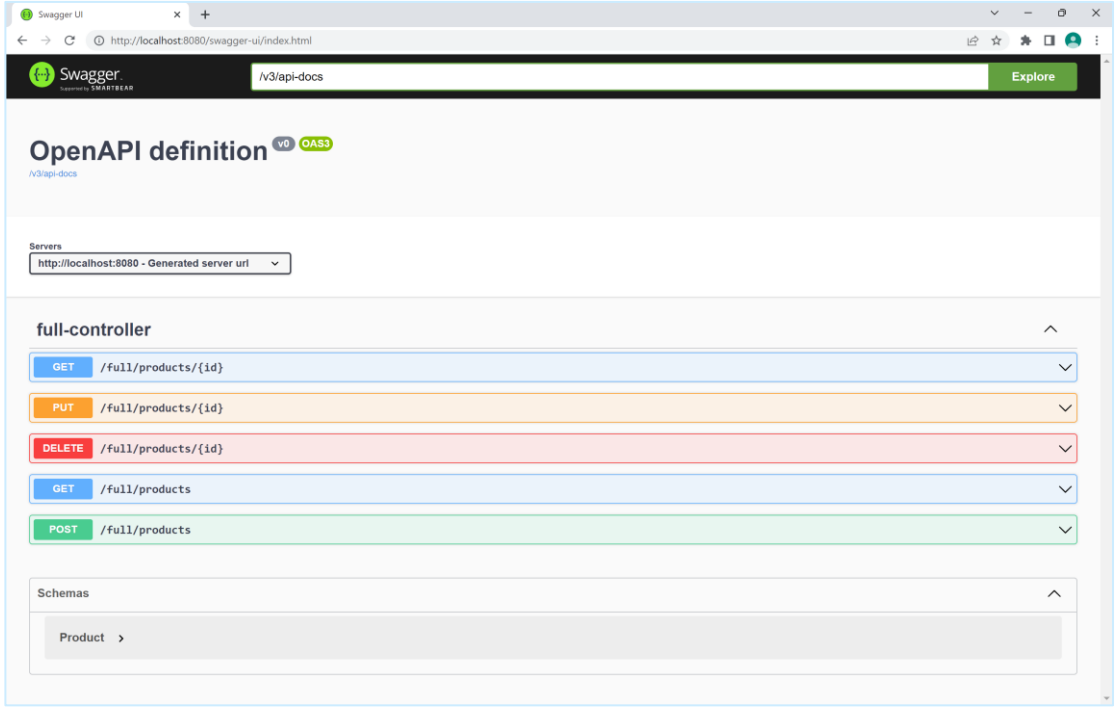
- To use Swagger in a Spring Boot application, add the following dependency to your POM file:

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

pom.xml

Using the Swagger UI

- Run your Spring Boot app and browse to:
  - http://localhost:8080/swagger-ui/index.html



2. Defining a Full REST Service

- Overview
- Implementing a DELETE method
- Implementing a PUT method
- Implementing a POST method

Overview

- In this section we'll see how to implement the following kinds of endpoints in a REST controller:
  - DELETE
  - PUT
  - POST

- We'll look at the DELETE endpoint first, because it's the most straightforward 😀

Implementing a DELETE Method

- A DELETE method typically deletes an existing resource:

```
@DeleteMapping("/products/{id}")
public ResponseEntity<Void> deleteProduct(@PathVariable long id) {

    if (!repository.delete(id))
        return ResponseEntity.notFound().build();
    else
        return ResponseEntity.ok().build();
}
```

FullController.java

- Client passes id in URL
- Service returns status code 200 or 404

Implementing a PUT Method

- A PUT method typically updates an existing resource:

```
@PutMapping("/products/{id}")
public ResponseEntity<Void> updateProduct(@PathVariable long id,
                                          @RequestBody Product product) {

    if (!repository.update(product))
        return ResponseEntity.notFound().build();
    else
        return ResponseEntity.ok().build();
}
```

FullController.java

  - Client passes id in URL
  - Client also passes an object in request body
  - Service returns status code 200 or 404

Implementing a POST Method

- A POST method typically inserts a resource:

```
@PostMapping("/products")
public ResponseEntity<Product> insertProduct(@RequestBody Product product) {

    repository.insert(product);
    URI uri = URI.create("/full/products/" + product.getId());
    return ResponseEntity.created(uri).body(product);
}
```

FullController.java

- Client passes object in HTTP request body
- Service returns enriched object after insertion
- Service also returns status code 201, plus a `LOCATION` header

Summary

- Setting the scene
- Defining a full REST service