Implementing a REST API

1. Getting started
2. Defining a simple REST API
3. Defining a full REST API

Demo Spring module: demo-rest-services

1. Getting Started

- Creating a Spring Boot web application
- The role of REST services
- REST services in Spring MVC
- Supporting JSON and XML
- Defining a model class

Demo package: `demo.restservices`

Creating a Spring Boot Web Application

- To create a Spring Boot web application, either add the Spring Web dependency…

We'll do this

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

pom.xml

- Or add the Spring Reactive Web dependency…
  - New in Spring Boot 2, good if you have very high load

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

pom.xml

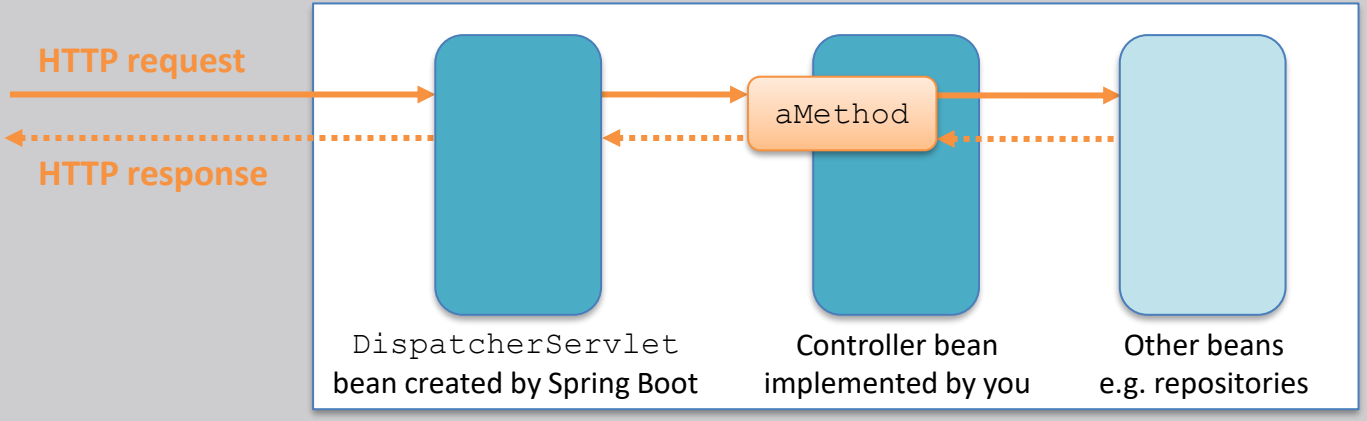
The Role of REST Services

- A REST service is an endpoint in a web application
  - Has methods that are mapped to URLs
  - Easily accessible by clients over HTTP(S)
  - Consume/return data, typically JSON (or XML)

- The role of REST services in a full-stack application:
  - Callable from UI, e.g. from a React web UI
  - Provides a façade to back-end data/functionality

REST Services in Spring MVC

- This is how REST services work in Spring MVC:
  - `DispatcherServlet` bean listens for HTTP requests
  - It dispatches a request to a method on a controller bean
  - The method returns data to the client



Supporting JSON and XML

- REST controller methods receive/return Java objects
- Spring Boot automatically creates a JSON serializer bean, to convert Java objects to/from JSON
- If you also want to support XML serialization, you must add the following dependency:

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

pom.xml

Defining a Model Class

- We'll use the following POJO class in our REST services

```
public class Product {
    private long id;
    private String description;
    private double price;

    // Plus constructors, getters/setters, etc …
}
```

Product.java

- The JSON/XML serializers will automatically convert `Product` objects to/from JSON/XML as appropriate

2. Defining a Simple REST API

- How to define a REST controller
- Example REST controller
- Pinging the simple REST controller
- A better approach
- Mapping path variables
- Mapping request parameters

How to Define a REST Controller

- Define a class and annotate with:
  - `@Controller` (or `@RestController`)
  - `@RequestMapping` (optional base URL)
  - `@CrossOrigin` (optional CORS support)
- Define methods annotated with one of the following:
  - `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@RequestMapping`
- For each method, also specify path and data-types

Example REST Controller

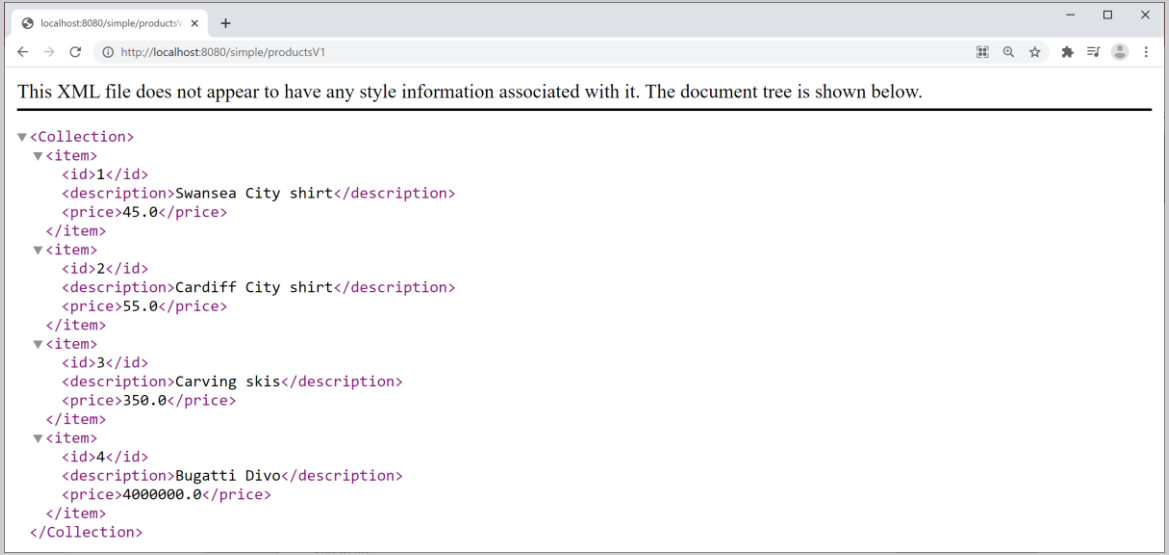
- Here's a simple REST controller
  - The method returns a collection of products

```
@RestController
@RequestMapping("/simple")
@CrossOrigin
public class SimpleController {

    private Map<Long, Product> catalog = new HashMap<>();
    …

    @GetMapping(value="/productsV1", produces={"application/json","application/xml"})
    public Collection<Product> getProductsV1() {
        return catalog.values();
    }
    …
}
```

SimpleController.java

Pinging the Simple REST Controller

- Run the Spring Boot app, then browse to:
  - http://localhost:8080/simple/productsV1



A Better Approach

- So far, we return a `Collection<Product>`
  - This populates the HTTP response body
  - It doesn't set the HTTP status code or any other headers
- A better approach is to return `ResponseEntity<T>`
  - Gives full control over the entire HTTP response body
  - Enables us to set HTTP status code and other headers

```
@GetMapping(value="/productsV2", produces={"application/json","application/xml"})
public ResponseEntity<Collection<Product>> getProductsV2() {
    return ResponseEntity.ok().body(catalog.values());
}
```

SimpleController.java

Mapping Path Variables

- You can map parts of the path to variables
  - In the path, define `{…}` placeholder(s)
  - In the method, annotate param with `@PathVariable`

```
http://localhost:8080/simple/products/1
```

```
@GetMapping(value="/products/{id}", produces={"application/json","application/xml"})
public ResponseEntity<Product> getProductById(@PathVariable long id) {

    Product p = catalog.get(id);
    if (p == null)
        return ResponseEntity.notFound().build();
    else
        return ResponseEntity.ok().body(p);
}
```

SimpleController.java

Mapping Request Parameters

- You can map HTTP request parameter(s)
  - In the path, optionally provide parameter(s) after ?
  - In the method, annotate param with `@RequestParam`

```
http://localhost:8080/simple/products?min=100
```

```
@GetMapping(value="/products", produces={"application/json","application/xml"})
public ResponseEntity<Collection<Product>> getProductsMoreThan(
            @RequestParam(value="min", required=false, defaultValue="0.0") double min) {

    Collection<Product> products = catalog.values()
                                          .stream()
                                          .filter(p -> p.getPrice() > min)
                                          .collect(Collectors.toList());
    return ResponseEntity.ok().body(products);
}
```

SimpleController.java

3. Defining a Full REST API

- Overview
- Example REST controller
- Testing the example REST controller
- Implementing a POST method
- Implementing a PUT method
- Implementing a DELETE method

Overview

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

```
@GetMapping(value= … )
```

- Here's how to support the other HTTP verbs

```
@PostMapping(value= … )
```

```
@PutMapping(value= … )
```

```
@DeleteMapping(value= … )
```

Example REST Controller

- Here's the example REST controller for this section:

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

    @Autowired
    private ProductRepository repository;

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

FullController.java

- Note:
  - We've now implemented a repository to manage data

Testing the Example REST Controller

- We'll test the service using ARC (a free Google plugin)
  - Install from https://install.advancedrestclient.com
- Allows you to submit all kinds of requests to a URL
  - GET, PUT, POST, DELETE, etc.
- Also allows you to set HTTP headers on your request
  - E.g. Content-Type=application/json
  - E.g. Accept=application/json

Implementing a POST Method

- A POST method typically inserts a resource
  - Client passes new object in HTTP request body
  - Service returns enriched object after insertion
  - Service returns status code 201, plus `LOCATION` header

```
@PostMapping(
        value="/products",
        consumes={"application/json","application/xml"},
        produces={"application/json","application/xml"})

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

Implementing a PUT Method

- A PUT method typically updates an existing resource
  - Client passes id in URL, and object in HTTP request body
  - Service returns status code 200 or 404

```
@PutMapping(value="/products/{id}", consumes={"application/json","application/xml"})
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

Implementing a DELETE Method

- A DELETE method typically deletes an existing resource
  - Client passes id in URL
  - Service returns status code 200 or 404

```
@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

Summary

- Getting started
- Defining a simple REST API
- Defining a full REST API