Implementing a Simple REST Service

1. Getting started
2. Defining a simple REST service

1. Getting Started

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

Demo package: `demo.restservices`

Spring Boot Web Applications

- To create a web app, add the Spring Web dependency:

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

pom.xml

We'll do this

- Alternatively, add the Spring Reactive Web dependency
  - Good if you have very high load or a continuous stream of data

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

pom.xml

Pearson

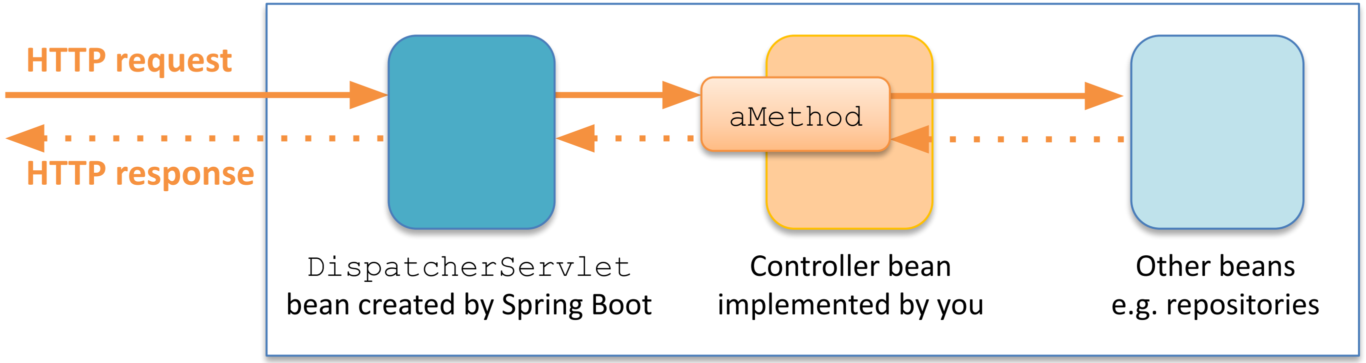
The Role of REST Services

- 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:



Spring Boot application

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 in your POM file:

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

pom.xml

Pearson

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 convert `Product` objects to/from JSON or XML automatically, as appropriate

Pearson

2. Defining a Simple REST Service

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

Pearson

Demo package: `demo.restservices`

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 the path (URL) and data-types

Example REST Controller

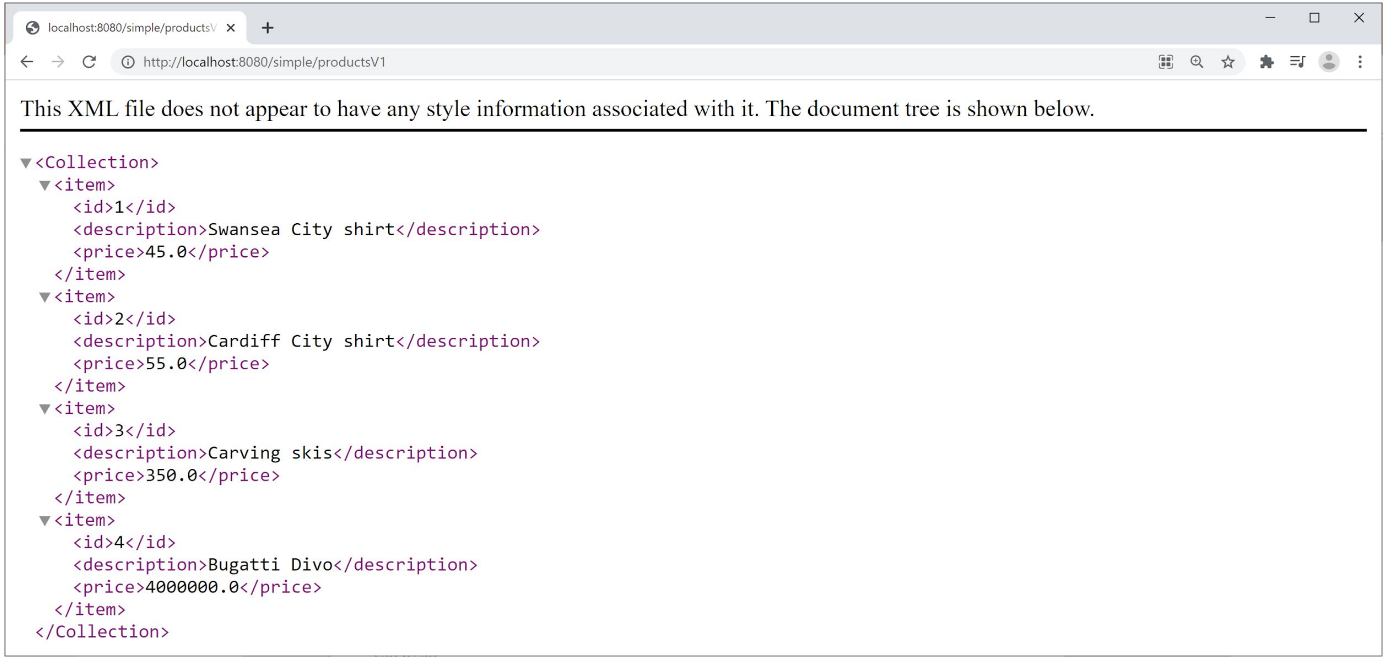
- Here's a simple REST controller
  - The method returns a product collection:

```
@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



localhost:8080/simple/productsV

http://localhost:8080/simple/productsV1

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<Collection>
  ▼<item>
     <id>1</id>
     <description>Swansea City shirt</description>
     <price>45.0</price>
   </item>
  ▼<item>
     <id>2</id>
     <description>Cardiff City shirt</description>
     <price>55.0</price>
   </item>
  ▼<item>
     <id>3</id>
     <description>Carving skis</description>
     <price>350.0</price>
   </item>
  ▼<item>
     <id>4</id>
     <description>Bugatti Divo</description>
     <price>4000000.0</price>
   </item>
 </Collection>
```

A Better Approach

- So far, we return a `Collection<Product>`
  - This populates the HTTP response body
  - But it doesn't set the HTTP headers or status code

- A better approach is to return `ResponseEntity<T>`
  - Gives control over entire HTTP response body
  - We can set HTTP headers and status code:

```
@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

```
@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

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

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`

```
@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
```

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

Pearson

Summary

- Getting started
- Defining a simple REST service

Exercise

- Add the following endpoints to the REST controller:
  - `GET  /count`
    Returns the count of products
  - `GET  /averagePrice?min=xxx&max=yyy`
    Returns the average price (in an optional range)