Integrating with Data Sources

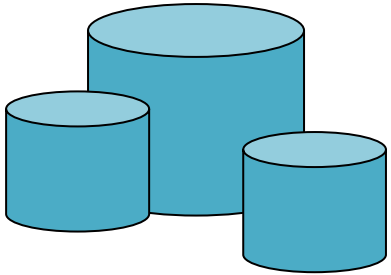
1. Understanding Spring Data
2. Getting started with JPA
3. Defining JPA entity classes
4. Viewing database data

1. Understanding Spring Data

- Spring vertical data access APIs
- About Spring Data
- Adding the data source driver to the classpath

Demo package: demo.springdata

Spring Vertical Data Access APIs



- Spring provides vertical APIs for data access
  - Many technologies, including JDBC, JPA, etc.
- Declarative transaction management
  - Transactional boundaries declared via configuration
  - Enforced by a Spring transaction manager
- Automatic connection management
  - Acquires/releases connections automatically

About Spring Data

- Spring Data supports many data access technologies
  - See https://spring.io/projects/spring-data
- Powerful repository and object-mapping abstractions
- Dynamic query creation from repository method names

Adding the Data Source Driver to the Classpath

- Add the appropriate Maven dependency for the type of data source you wish to access, e.g. H2:

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

pom.xml

- H2 is an in-memory database
  - Created/dropped when app starts/ends
  - Very handy during development ☺

2. Getting Started with JPA

- Overview of JPA
- Important JPA concepts
- JPA dependency in Spring Boot
- Spring Boot autoconfiguration
- Customizing persistence properties

Overview of JPA

- JPA = Java Persistence API
  - A standard ORM (object/relational mapping) API
- JPA is a specification
  - Implemented by the Hibernate library
  - Also implemented by Java Enterprise Edition
- To use JPA in Spring:
  - Add the Hibernate library to your classpath, see later

Important JPA Concepts

- Entity class
  - A Java class, mapped to a relational database table
- Entity manager
  - Provides an API to fetch/save entities to a relational database
- Entity manager factory
  - Creates and configures an entity manager so it can connect to a relational database

JPA Dependency in Spring Boot

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

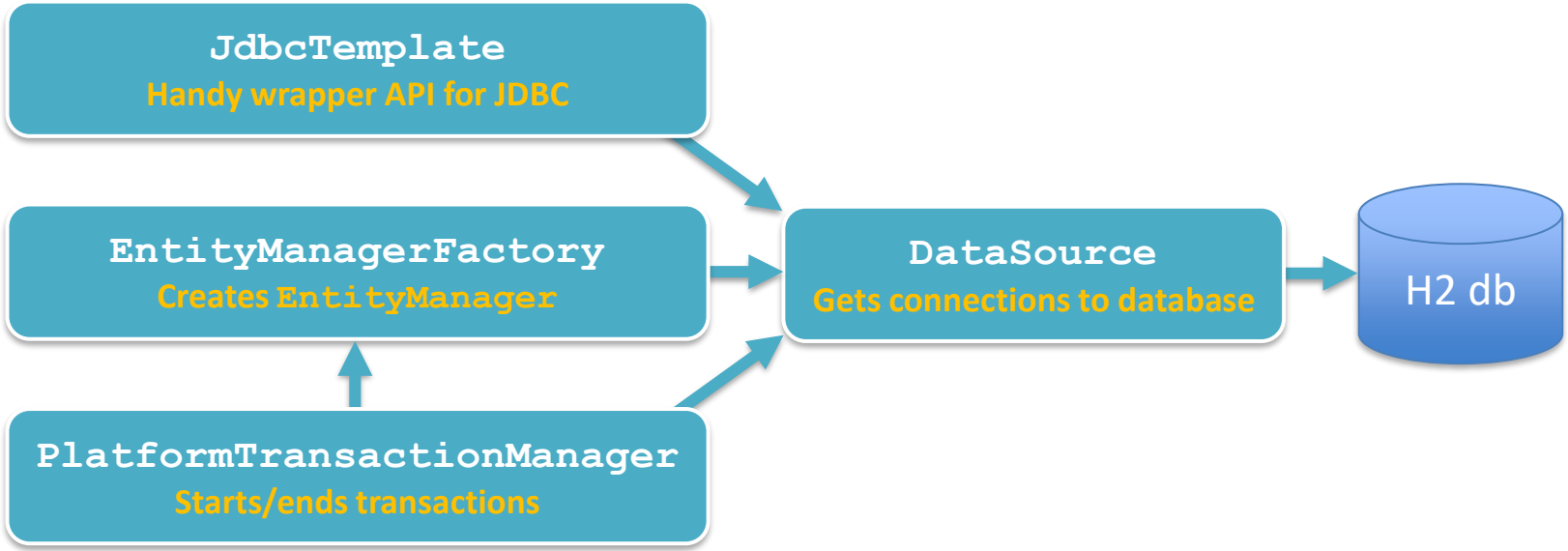
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

pom.xml

- This adds all the relevant Hibernate libraries to the classpath

Spring Boot Autoconfiguration

- Courtesy of the JPA dependency, Spring Boot creates several beans automatically in your application



Customizing Persistence Properties

- Spring Boot automatically sets persistence properties to connect to the in-memory H2 database:

```
spring.datasource.url=jdbc:h2:mem:<UUID>
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

application.properties

- You can customize persistence properties if you need to:

```
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.use_sql_comments=true
spring.jpa.properties.hibernate.format_sql=true
```

application.properties

3. Defining JPA Entity Classes

- How to define an entity class
- Locating entity classes
- Seeding the database with data

How to Define an Entity Class

- You can define an entity class as follows:

```
import javax.persistence.*;

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

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long employeeId = -1;

    private String name;
    private String region;

    @Column(name="salary")
    private double dosh;

    // Plus constructors, getters/setters,
    // equals(), and hashCode()
}
```

Employee.java

Locating Entity Classes

- A Spring Boot app scans for entity classes when it starts
  - It looks in the main app class package, plus sub-packages
- You can tell it to look elsewhere, if necessary
  - Via `@EntityScan`

```
@SpringBootApplication
@EntityScan( {"myentitypackage1", "myentitypackage2"} )
public class Application {
    …
}
```

Seeding the Database with Data

- For convenience during development/testing, you can seed the database with some sample data

```
import org.springframework.jdbc.core.JdbcTemplate;
…

@Component
public class SeedDb {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @PostConstruct
    public void init() {
        jdbcTemplate.update(
            "insert into EMPLOYEES(name,salary,region) values(?,?,?)",
            new Object[]{"James", 21000, "London"});
        …
    }
}
```

SeedDb.java

4. Viewing Database Data

- Overview
- Obtaining the database connection string
- Viewing the database data in the H2 console UI

Overview

- Most databases have a console UI to let you view data
  - To enable the H2 console UI, add these application properties:

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

application.properties

- The H2 console UI is a web endpoint
  - So, add this dependency in your POM:

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

pom.xml

Obtaining the Database Connection String

- When you run your app, you'll see a message that indicates the JDBC connection string for the database:

```
.kariDataSource       : HikariPool-1 - Start completed.
.AutoConfiguration    : H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:d58eb18c-b573-4967-a6e2-ce52b628e561'
.rnal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
.n                    : HHH000412: Hibernate ORM core version 5.4.28.Final
.ons.common.Version   : HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
```

- You can use this JDBC connection string to connect to the database in the H2 console UI …

Viewing the Database Data in the H2 Console UI

- To open the H2 console UI, browse to:
  - http://localhost:8080/h2-console

- To connect to the database, enter these details:
  - JDBC URL - as per previous slide
  - User name - `sa`
  - Password - leave blank

- You can then view tables in the database - cool!

Summary

- Understanding Spring Data
- Getting started with JPA
- Defining JPA entity classes
- Viewing database data

Exercise

- Define an entity class named `Car` with these fields:
  - `carId` (primary key)
  - `registrationNumber`
  - `make`
  - `model`
- Add some code in `SeedDb` to insert some cars
- Run the application and view the H2 console UI, to confirm the car data exists in the database

Querying and Modifying Entities

1. Querying entities
2. Modifying entities

1. Querying Entities

- Defining a repository component
- Finding an entity by primary key
- Working with queries
- Performing a simple query
- Getting a list of entities

Demo package: demo.springdata

Defining a Repository Component

- In this chapter we'll use JPA to query and modify entities
  - Via methods in the JPA `EntityManager` class

- We'll put all our data-access code in a repository component
  - We'll inject an `EntityManager` bean as follows:

```
import javax.persistence.*;
…

@Repository
public class EmployeeRepository {

    @PersistenceContext
    private EntityManager entityManager;

    // Methods to create, read, update, delete entities.
    …
}
```

EmployeeRepository.java

Finding an Entity by Primary Key

- To find an entity by primary key:
  - Call `find()` on the `EntityManager`
  - Returns `null` if entity not found

```
public Employee getEmployee(long employeeId) {
    return entityManager.find(Employee.class, employeeId);
}
```

EmployeeRepository.java

Working with Queries

- Define a query string
  - Using JPQL (or SQL)
- Create a `TypedQuery<T>` object
  - Via `createQuery()` on the `EntityManager`
- Execute the query via one of these methods:
  - `getSingleResult()`
  - `getResultList()`

Performing a Simple Query

- Here's a query that returns a single result:

```
public long getEmployeeCount() {
    String jpql = "select count(e) from Employee e";
    TypedQuery<Long> query = entityManager.createQuery(jpql, Long.class);
    return query.getSingleResult();
}
```

EmployeeRepository.java

Getting a List of Entities

- Here's a query that returns a list of entities:

```
public List<Employee> getEmployees() {
    String jpql = "select e from Employee e";
    TypedQuery<Employee> query = entityManager.createQuery(jpql, Employee.class);
    return query.getResultList();
}
```

EmployeeRepository.java

2. Modifying Entities

- Overview
- Inserting an entity
- Updating an entity
- Deleting an entity

Overview

- JPA lets you insert, update, and delete entities

- You must put these operations in a transactional method in a component class
  - Annotate method with `@Transactional`

```
@Transactional
public void someMethodToModifyEntities() {
    …
}
```

Inserting an Entity

- This is how you insert an entity in the database:

```
@Transactional
public void insertEmployee(Employee e) {
    entityManager.persist(e);
}
```

EmployeeRepository.java

Updating an Entity

- This is how you update an entity in the database:

```
@Transactional
public void employeePayRise(long id, double payRise) {
    Employee emp = entityManager.find(Employee.class, id);
    emp.setDosh((emp.getDosh() + payRise));
}
```

EmployeeRepository.java

Deleting an Entity

- This is how you delete an entity in the database:

```
@Transactional
public void deleteEmployee(long employeeId) {
    Employee e = entityManager.find(Employee.class, employeeId);
    entityManager.remove(e);
}
```

EmployeeRepository.java

Summary

- Querying entities
- Modifying entities

Exercise

- Add a method in `EmployeeRepository`, to give a pay rise to all employees in a region, as follows:

1. Define a parameterized JPQL query string:

```
String q = "update Employee set dosh=dosh+:p " +
           "where region=:r";
```

2. Create a query and set parameters on it:

```
Query query = entityManager.createQuery(q);
query.setParameter("p", payRise);
query.setParameter("r", region);
```

3. Execute the query as an "update" statement:

```
int numRowsAffected = query.executeUpdate();
```

Spring Data Repositories

1. Understanding Spring Data repositories
2. Using a Spring Data repository

1. Understanding Spring Data Repositories

- Overview of Spring Data repositories
- Spring Data repository capabilities
- Paging and sorting
- Technology-specific repositories
- Domain-specific repositories

Overview of Spring Data Repositories

- Spring Data is a data-access abstraction mechanism
  - Makes it very easy to access a wide range of data stores
  - Using a familiar "repository" pattern
  - Create / Read / Update / Delete (CRUD)

- It provides template repositories for…
  - JPA
  - MongoDB, Cassandra, CouchBase
  - Etc.

Spring Data Repository Capabilities

- Spring Data defines a general-purpose repository interface:

```
public interface CrudRepository<T,ID> {

    long count();

    void delete(T entity);

    void deleteAll();

    void deleteAll(Iterable<T> entities);

    void deleteById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    Optional<T> findById(ID id);

    T save(T entity);

    Iterable<T> saveAll(Iterable<T> entities);
}
```

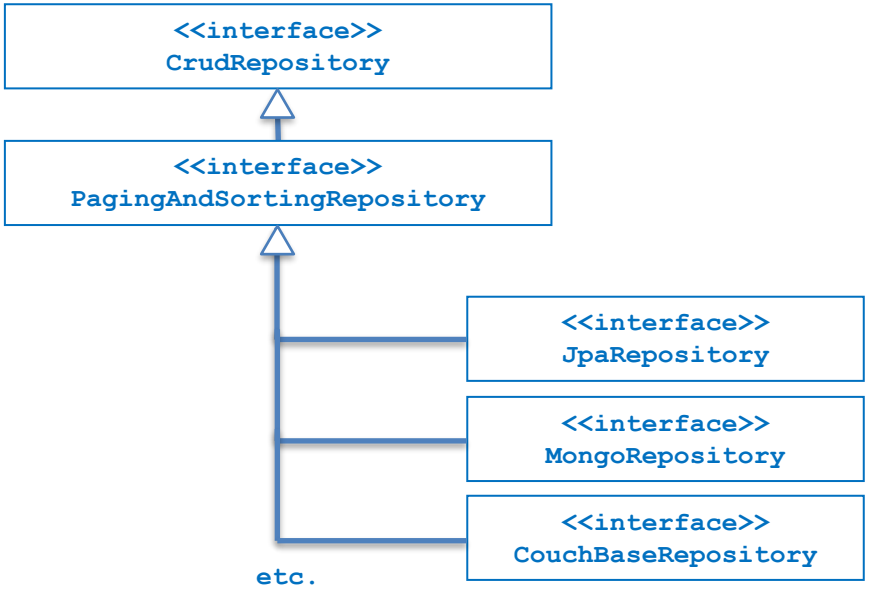
Paging and Sorting

- Support for paging and sorting is provided via this interface:

```
public interface PagingAndSortingRepository<T,ID>
       extends CrudRepository<T,ID> {

    Page<T> findAll(Pageable pageable);

    Iterable<T> findAll(Sort sort);
}
```

Technology-Specific Repositories

- Spring Data also provides technology-specific repositories
  - Interfaces that extend `PagingAndSortingRepository`
  - Provide technology-specific extensions



Domain-Specific Repositories

- You can define your own domain-specific interfaces
  - Extend `CrudRepository` (or sub-interface)
  - Specify the entity type and the PK type
- You can define specific query methods for your entities
  - Spring Data reflects on method names to create queries
  - You can provide an explicit query string for complex queries
- See next section for an example…

2. Using a Spring Data Repository

- Overview
- Defining a repository
- Locating Spring Data repositories
- Using a Spring Data repository

Demo package: demo.springdatarepositories

Overview

- In this section we'll see how to access a relational database by using a Spring Data repository

- Note the following key points in the demo first:
  - `pom.xml`
  - `application.properties`
  - `Employee.java`
  - `SeedDb.java`

Defining a Repository

- Here's an example of a domain-specific repository:

```
public interface EmployeeRepository extends CrudRepository<Employee, Long> {

    List<Employee> findByRegion(String region);

    @Query("select e from Employee e where e.dosh >= ?1 and e.dosh <= ?2")
    List<Employee> findInSalaryRange(double from, double to);

    Page<Employee> findByDoshGreaterThan(double salary, Pageable pageable);

}
```

EmployeeRepository.java

- Note:
  - Entity type is `Employee`, PK type is `Long`
  - Also, we've defined some custom queries

Locating Spring Data Repositories

- A Spring Boot application scans for Spring Data JPA repository interfaces when it starts
  - It looks in the main application class package, plus sub-packages

- You can tell it to look elsewhere, if you like
  - Via `@EnableJpaRepositories`

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
…

@SpringBootApplication
@EnableJpaRepositories({"repopackage1", "repopackage2"})
public class Application {
    …
}
```

Using a Spring Data Repository (1 of 2)

- Let's see how to use some standard repository methods:

```
@Component
public class EmployeeService {

    @Autowired
    private EmployeeRepository repository;

    public void useStandardRepoMethods() {

        // Insert an employee.
        Employee newEmp = new Employee(-1, "Simon Peter", 10000, "Israel");
        newEmp = repository.save(newEmp);
        System.out.printf("Inserted employee, id %d\n", newEmp.getEmployeeId());

        // Get count of all employees.
        System.out.printf("There are now %d employees\n", repository.count());

        // Get all employees.
        displayEmployees("All employees: ", repository.findAll());
    }
    …
}
```

EmployeeService.java

Using a Spring Data Repository (2 of 2)

- Let's see how to use our custom queries in the repository:

```
@Component
public class EmployeeService {

    @Autowired
    private EmployeeRepository repository;

    public void useCustomQueryMethods() {

        // Get all employees by region.
        displayEmployees("All employees in London: ", repository.findByRegion("London"));

        // Get employees by salary range.
        List<Employee> emps = repository.findInSalaryRange(20000,  50000);
        displayEmployees("Employees earning 20k to 50k: ", emps);

        // Get a page of employees.
        Pageable pageable = PageRequest.of(1, 3, Direction.DESC, "dosh");
        Page<Employee> page = repository.findByDoshGreaterThan(50000, pageable);
        displayEmployees("Page 1 of employees more than 50k: ", page.getContent());
    }
}
```

EmployeeService.java

Summary

- Understanding Spring Data repositories
- Using a Spring Data repository

Exercise

- We've seen how to define a custom "select" method
  - Annotate a method with `@Query`
  - Specify a "select" JPQL string
- It's also possible to define a custom "modifying" method
  - Annotate with `@Query`, `@Modifying`, `@Transactional`
  - Specify an "insert", "update", or "delete" JPQL string

```
public interface EmployeeRepository extends CrudRepository<Employee, Long> {

    @Modifying(clearAutomatically=true)
    @Transactional
    @Query("delete from Employee e where e.dosh >= ?1 and e.dosh <= ?2")
    int deleteInSalaryRange(double from, double to);
    …
}
```

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

- 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

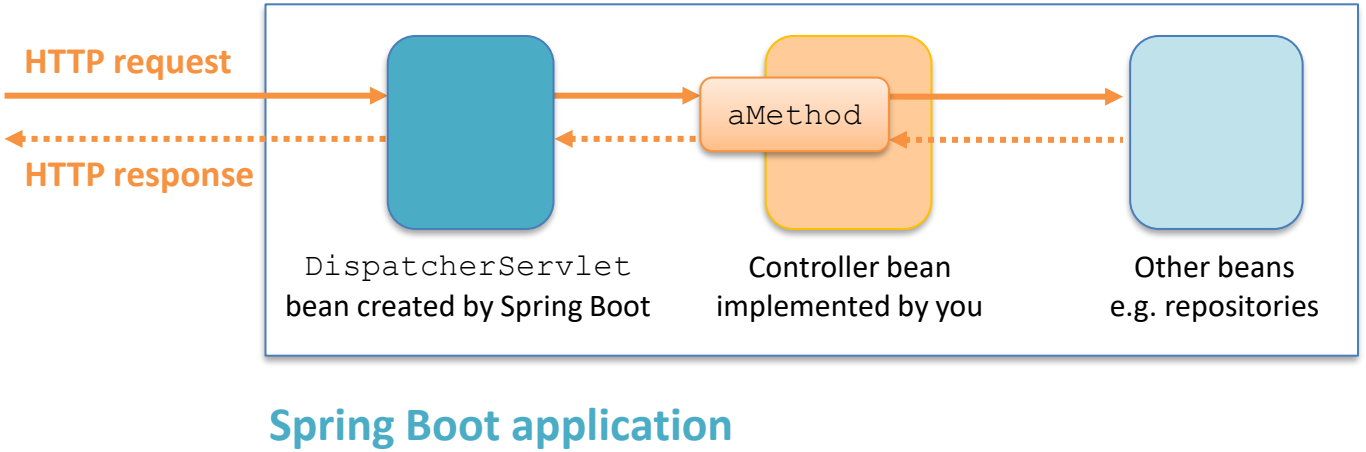
The Role of REST Services

- 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:



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

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

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

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

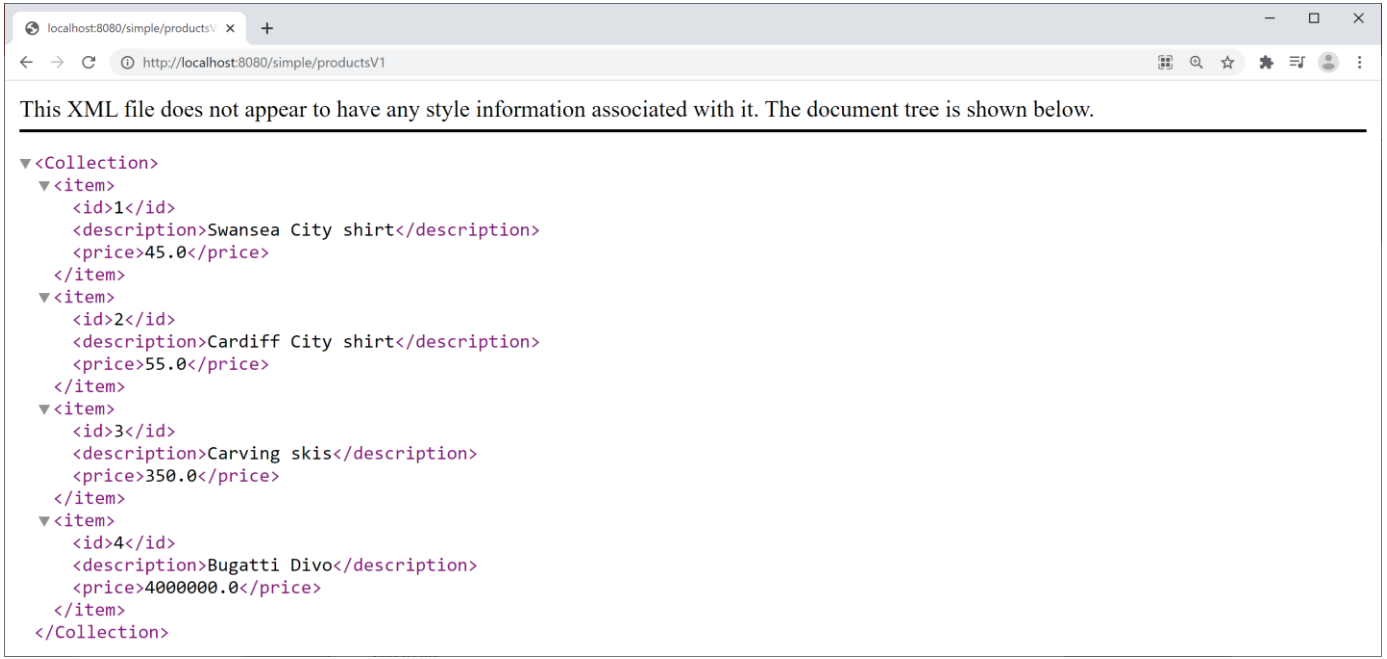
- Here's a simple REST controller
  - The method returns a product collection:

```
@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



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
```

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)

Implementing a Full REST Service

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

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(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 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 3)

- 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 Swagger, see:
  - https://swagger.io/

Using Swagger to Expose the REST API (2 of 3)

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

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>3.0.0</version>
</dependency>
```

pom.xml

Using Swagger to Expose the REST API (3 of 3)

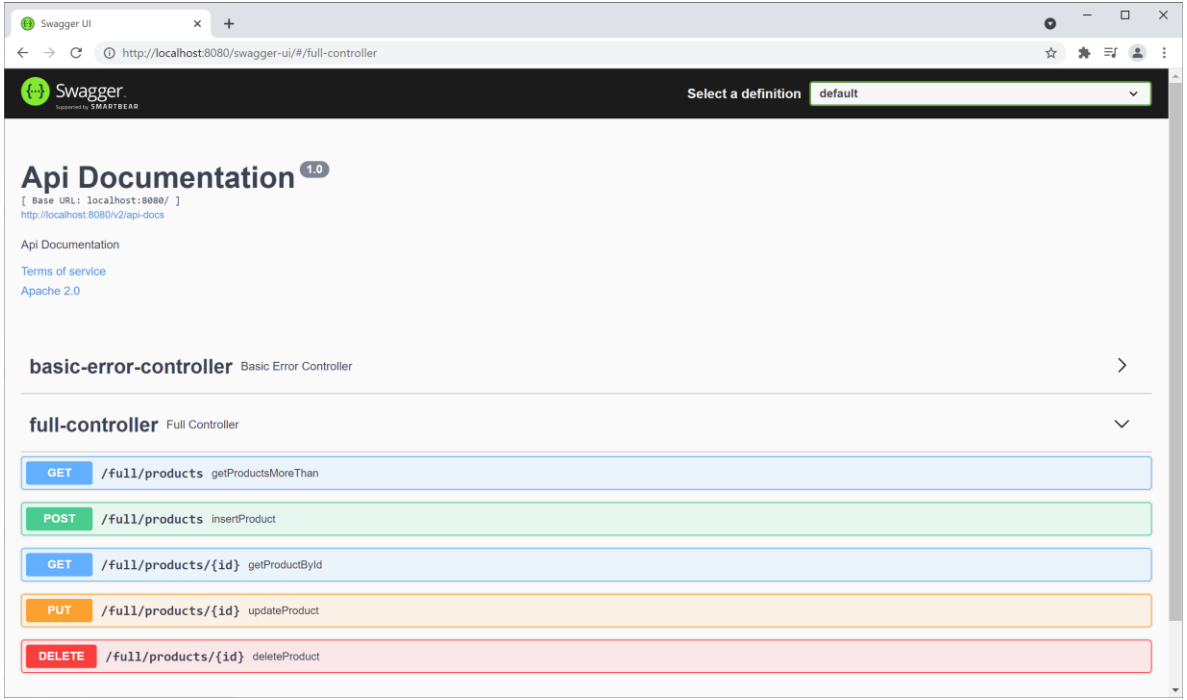
- You must also configure Swagger so it knows what controllers and paths to expose:

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

@Configuration
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(PathSelectors.any())
                .build();
    }
}
```

SwaggerConfiguration.java

Using the Swagger UI

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



2. Defining a Full REST Service

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

Implementing a POST Method

- A POST method typically inserts a resource:

```
@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

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

Implementing a PUT Method

- A PUT method typically updates an existing resource:

```
@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

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

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

Summary

- Setting the scene
- Defining a full REST service

Exercise

Add the following endpoint to the REST controller:

- `PUT  /products/1/increasePriceBy/10.99`
  Increases price of specified product by specified amount

Consuming REST Services

- Overview
- Key methods in RestTemplate
- Example
- Key classes in the REST client application
- Aside: Consuming a REST service from HTML

Overview

- Spring enables you to implement client code to consume REST services
  - Via the `RestTemplate` class

- Include the following POM dependency:

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

pom.xml

Key Methods in RestTemplate

- Here are some of the key methods in `RestTemplate`:

```
ResponseEntity<T> getForEntity(String, Class<T>, Object...)
```

```
ResponseEntity<T> postForEntity(String, Object, Class<T>, Object...)
```

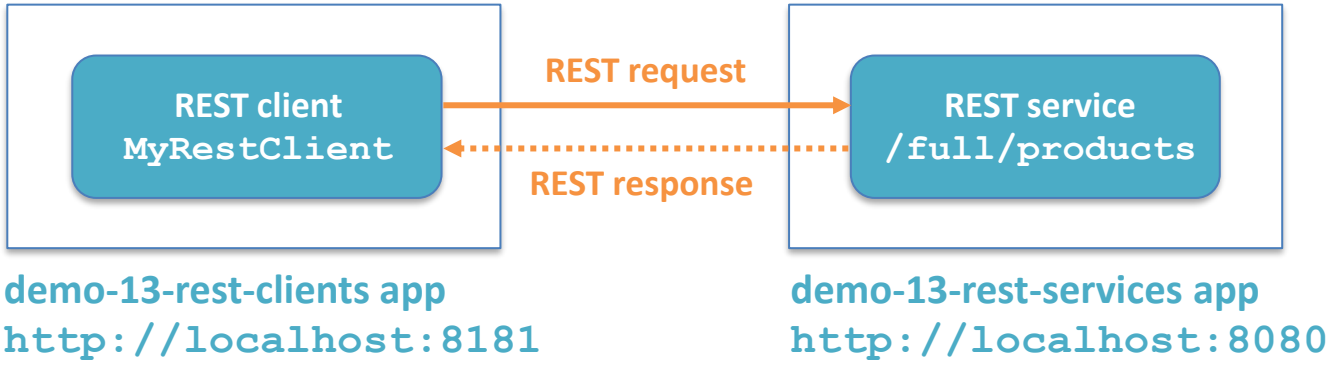
```
void put(String, Object, Object...)
```

```
void delete(String, Object...)
```

```
ResponseEntity<T> exchange(String, HttpMethod, object, Class<T>)
```

Example

- Let's see an example of how to consume REST endpoints:



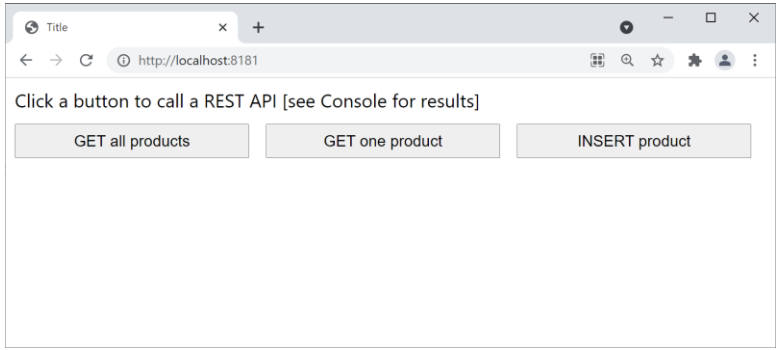
Key Classes in the REST Client Application

- `MyRestClient`
  - Calls REST service endpoints, by using `RestTemplate`
- `Product`
  - `Product` objects passed to/from REST service
  - Serialized/deserialized by `RestTemplate`

Aside: Consuming a REST service from HTML

- We've also implemented a simple HTML page to show how to consume a REST service from a web UI
  - Project: `demo-13-rest-clients`
  - Folder: `src/main/resources/static`

- Open a browser and browse to `http://localhost:8181`



Summary

- Overview
- Key methods in RestTemplate
- Example
- Key classes in the REST client application
- Aside: Consuming a REST service from HTML