**Intro**

Via the URL <https://spring.io/guides/> you can find numerous getting started guides.

For reference documentation you can navigate to <http://docs.spring.io/spring-boot/docs/1.1.x/reference/htmlsingle/>

**Lab 2**

**Create a new REST service with Spring Boot**

In lab 1 you made your first REST service in Java with the Springboot framework (if not you can use the completed services that is included with the first lab).

In this lab you are creating your second REST service, named ProductService, and make some changes to the ShoppingService so that it calls the ProductService for its list of drinks. Along you will also make your first use of the Spring autowiring feature.

**Create your second project**

In the first lab, you were introduced to the web based Spring Initializr, which generates an ‘out of the box’ project structure for you. For the creation of your second REST service you can make use of it again or use an embedded/plugin version in the IDE of your choice.

* Name the Artifact ‘productservice’
* Fill in ‘demo project for Spring Boot’ for the description
* Change package name to: nl.groothandel.service
* Set the packaging option to jar
* Choose java version 8 (or higher, depending on your installed JDK version)
* Choose WEB as a dependency
* If not already so, open the project in your favorite IDE

**Add a JSON Controller**

* Add a new folder to the package ‘nl.groothandel.service’ and name it ‘web’
* Add a new Java class to the web folder and name it ‘ProductsController.java’
* Annotate the class definition as a REST controller and give it a request mapping with the value ‘products’ (see lab 1 for more details, if you are not sure how to do this). By this request mapping annotation the products controller is reacheable via the URL <http://[servername]:[port]/products>.

**Add a domain with products**

In the shopping service (lab 1), drinks were retrieved from a hard coded list of String values. The ProductService however, will be serving drinks as a list of Products. For now these products will still be hard coded, but later on in the labs (in the 2nd SIG) these products will be retrieved from a database.

For now, you are going to add 2 domain classes to the productservice.

* Add a new folder to the package ‘nl.groothandel.service’ and name it ’domain’
* Add a new Java class to the domain folder and name it Product.java
* Add another Java class to the domain folder and name it ProductRepositoryImpl.java
* Open the Product.java class and add the following properties to the class definition:

private String productId;

private String name;

private String type;

private String brand;

private String bottleSize;

private double abv; //eg. 20 pct alcohol

private double price;

* Also add the following 2 constructor methods to the Product.java class. A constructor method is called upon automatically at runtime during instantiation.

public Product ();

public Product (String productId, String name, String type, String brand, String bottleSize, double abv, double price) {

super();

this.productId = productId;

this.name = name;

this.type=type;

this.brand=brand;

this.bottleSize=bottleSize;

this.abv=abv;

this.price=price;

}

* Generate or make getter en setter methods for each property

public String getProductId() {

return productId;

}

public void setProductId(String productId) {

this.productId = productId;

}

***…(repeat this for other properties)***

* Open the ProductRepositoryImpl.java class and add a collection (java.util.Map) of drinks to it (just below line with the class definition):

private Map<String, Product> products;

{

Map<String, Product> map = new HashMap<>();

map.put(“GLNF”, new Product(“GLNF”,”Glenfiddich 18 Years”, “Whisky”, “Glenfiddich”, “70cl”, 42, 39.95);

map.put(“PULT”, new Product(“PULT”,” Old Pulteney 17 years Single Malt”, “Whisky”, “Old Pulteney”, “70cl”, 41, 59.95);

this.products = map;

}

In lab 1 you added a hard coded list of drinks to the shoppingService. By the following instructions in this lab, the shopping service will retrieve its list of drinks from the product service and so, no so longer make use of the embedded hardcoded list. Therefor it will be removed from the shopping service later on.

* For now, add a getter and setter method for the map to the ProductRepositoryImpl.java class

public Map<String, Product> getProducts() {

return products;

}

public void setProducts(Map<String, Product> products) {

this.products = products;

}

For now, this completes your domain definition.

* Open the file ProductsController.java

Below the line with the class definition add the following :

ProductRepositoryImpl productRepository = new ProductRepositoryImpl();

As you probably know, this is a so called ‘hard wired dependency’ also known as ‘tight coupling’ and can have unwanted implications during the classes life cycle. This line is coded as is for learning purposes. You are going to make changes to it later.

* Add the following method to the file ProductsController.java for retrieving a list of products. This list contains the products that you added earlier to the map in the ProductRepositoryImpl.java class.

public List<Product> retrieveProducts() {

return new ArrayList<>(productRepository.getProducts().values());

}

* The retrieveProducts() method should be invokeable via http GET and the URL <http://[servername]:[port]/products/>, which you already set for the class level. Add an appropriate annotation to the method. If you are not sure about the annotation, have a look at the finished code which comes with this lab.
* Add a second method to the controller class for getting a specific product from the list via the url: [http://[localhost]:[port]/products/{productId}](http://[localhost]:[port]/products/%7bproductId%7d). Again, can you figure out the annotation?

public Product retrieveProduct (@PathVariable(“productId”) String productId) {

return productRepository.getProducts().get(productId);

}

**Change the portnumber for the web server**

* Open the file application.properties, which already exists in your project folder, and add the following line:

Server.port=8081

Since the shopping service already runs on port 8080, another port is needed for the product service

**Test the ProductService**

For testing purposes you can use Postman again.

* Choose GET
* Type the URL : http:[server]:[port]/products with the placeholders [server] and [port] replaced with your actual values: likely localhost and 8081.
* Press send.
* If all goes well, then the list with drinks is returned

**Adjust the ShoppingService**

Now you have the product service returning a list op products in JSON format, it is time to make adjustments to the shopping service for using this list, (instead of the hard coded list in the shopping service).

* Reopen the ShoppingService in case you closed this.
* In the file ShoppingController.java adjust the method retrieveDrinks:
  + (Temporarily) adjust the method line to:

public static List retrieveDrinks()

Since the list from the product service contains products, and no longer Strings, the generic String has to be removed from List<String>. The change to just List allows the method to return any object type. Further on you are going to restrict the list to return only products.

* + Replace the line with the statement ‘return drinks;’ with the following:

RestTemplate restTemplate = new RestTemplate();

return restTemplate.getForObject(“<http://localhost:8081/products>”, List.class);

* Test the shopping service: <http://[server]:[port]/shopping/drinks> (GET). A json list with the drinks retrieved from the productservice should be in the result.
* Add a new ‘domain’ folder to the package nl.sjop.service and create a new class Product.java into it. The contents of the Product.java class is identical to the one in the product service (except for the package name).
* In the ShoppingController class now change List to List<Product> :

public static List<Product> retrieveDrinks()

* Check everything still works by testing it again.

The method retrieveDrink still retrieves its data from within, so let’s change that as well.

* Change the line with the method declaration retrieveDrink into:

public static **Product** retrieveDrink(@PathVariable(“id”) String productId)

Note the changed return type, which is now Product

* Replace the content of the method with the following code:

final String url = ”[http://localhost:8081/products/{productId}](http://localhost:8081/products/%7bproductId%7d)”;

Map<String, String> uriParams = new HashMap<>();

uriParams.put(“productId”, productId);

UriComponentsBuilder builder = UriComponentsBuilder.fromUriString(url);

URI uri = builder.buildAndExpand(uriParams).toUri();

//

RestTemplate restTemplate = new RestTemplate();

return restTemplate.getForObject(uri, **Product.class**);

The builder in this code creates an URI with the productId replaced with the value the method receives in the parameter productId. Again, this value is passed to the method via the ShoppingController when it receives a request via the url: [http://[server]:[port]/shopping/drink/{id}](http://[server]:[port]/shopping/drink/%7bid%7d), with server, port and id replaced with actual values. The created uri is passed to the method getForObject of the RestTemplate class. In turn it receives an instance of Product class, which can be used for further processing.

Note that this rest call invokes the method retrieveProduct in the product service over http.

* Remove the hard coded list named drinks from the ShoppingController.java class.

**Test the ShoppingService**

Test1: retrieve the list with drinks by invoking: <http://[server]:[port]/shopping/drinks>

Test2: retrieve the list with drinks by invoking: [http://[server]:[port]/shopping/drink/{id}](http://[server]:[port]/shopping/drink/%7bid%7d)

Eg. <http://localhost:8080/shopping/drink/PULT> or <http://localhost:8080/shopping/drink/GLNF>

**HTTP Status**

It might be possible to enter a productid which does not exist. As of now, the ProductService just returns NULL (productRepository.getProducts().get(productId); returns NULL if the productId is not found). Let’s change the code to return a ResponseEntity which contains the correct HTTP status code.

* Open in the ProductService the class ProductController.java
* Change the method retrieveProduct to:

public **ResponseEntity<Product>** retrieveProduct (@PathVariable("productId") String productId) {

if (productRepository.getProducts().containsKey(productId)) {

return new ResponseEntity<>(productRepository.getProducts().get(productId), HttpStatus.OK);

} else {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

Notice that the method signature has changed. The method will return a ResponseEntity containing a Product. Also, the method checks if the requested product exists and returns it along with a HTTP status code 200 if it exists. Otherwise only the HTTP status code 404 is returned.

* Test the ProductService with an existing and a non-existing product. Check the http status of both responses.

Now we can alter the ShoppingService to handle the 200/404 returned by the ProductService.

* Open the ShoppingService again
* Change the content of the retrieveDrink method:

final String url = ”[http://localhost:8081/products/{productId}](http://localhost:8081/products/%7bproductId%7d)”;

Map<String, String> uriParams = new HashMap<>();

uriParams.put(“productId”, productId);

//

RestTemplate restTemplate = new RestTemplate();

ResponseEntity resp = restTemplate.**exchange**(uri, HttpMethod.GET, Product.class, uriParams);

if (HttpStatus.OK.equals(resp.getStatusCode())) {

return resp.getBody();

} else {

return new Product();

}

Note that we are using exchange now instead of getForObject. Also, we are passing the uriParams as extra parameter instead of encapsulating it in the url.

* Next, alter the retrieveDrink method to also return a ResponseEntity<Product> in the same way as the ProductService. Return a http status code 200 if the product is found, otherwise a 400.
* Test the ShoppingService and see if you get the correct http status codes for existing and non-existing products.

**External API**

The shop also needs to show the price in USD. This can be hardcoded for each product, but it is better to call another API to retrieve the exchange rate between EUR and USD and use that rate to calculate the price in USD. For this purpose, we are gonna use the API exposed at <https://api.exchangeratesapi.io/>. This API will return exchange rates from a currency to other currencies.

* Open the ShoppingService and add an extra instance field to the Product.java class:

private double priceUSD;

* And add the getter/setter:

public String getPriceUSD () {

return priceUSD;

}

public void setPriceUSD(String priceUSD) {

this.priceUSD = priceUSD;

}

* Look at the response from <https://api.exchangeratesapi.io/latest?base=EUR>. The JSON response contains the base name, date of the exchange rate and a list of exchange rates.
* Create a Java class with the name Currency in the model folder and add an instance field for each property in the JSON response. Tip: use Map<String, Double> as type for the rates.
* Next, create/generate getters and setters for these fields.
* Inside the retrieveDrink method of the ShoppingController class, add a call to <https://api.exchangeratesapi.io/latest?base=EUR> using the RestTemplate class. Mind the use of the request parameter (everything behind the ? in an url is request parameters). These need to be set in a Map which is used as input for the RestTemplate.
* Calculate the price in USD using the exchange rate and store it in the priceUSD variable of the Product object. Use the following piece of code to get the exchange rate in USD from the response:

double usd = Double.parseDouble(currency.getRates().get("USD"));

* Test the ShoppingService again and see if you get a price in USD for a product.