**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 3**

**Post method**

In lab 2 you made a new product service for returning one or more drinks from a hard coded list. You also made some changes to the shopping services, which you created earlier in lab 1, to retrieve the data from the product service over http.

In this lab you are going to extend the services with methods for adding and updating the list of drinks.

Adding or creating a new entity is done by an http POST, which is a NON-idempotent method. This means that the result of a second invoke of the same requests changes the outcome, which is another 2nd occurrence added. Data to be added is posted in the body of the http request.

Contrarily Http GET, OPTIONS an PUT are (or should be implemented as) idempotent methods, which means that no matter how many invokes are done, the result does not change. For example , if an update request for a name field is repeated, the name field will still hold this updated value.

**Extend your product service**

* Open your product service from lab 2, if it is closed.
* Open the file ProductRepistoryImpl.java and add a method for adding a product to the products map.
  + Name the method addProduct
  + The method receieves 2 parameters: first String id, second Product product
  + Return a boolean value, indicating if the product is added to the map
* In the file ProductsController.java add a new method addProduct for adding a new drink product to the list. The method will receive a productId from the url, and the data for the product in the request body. Also an UriComponentsBuilder is passed along by Spring injection, which can be used for building encoded Uri components. In the response returned from this method, the persisted product will be returned.

public ResponseEntity<Product> addProduct(

@PathVariable(“productId”) String productId,

**@RequestBody** Product product,

UriComponentsBuilder builder ) {

*[code to be implemented]*

}

* Annotate this method with the same request mapping as the one for the retrieveProduct method, which should already be present in the file, and change the text RequestMethod.GET to RequestMethod.POST.

**Note!** The url [http://[localhost]:[port]/products/{productId}](http://[localhost]:[port]/products/%7bproductId%7d) is used here to invoke the method retrieveProduct and the method addDrink as well. Which of these methods is invoked, depends on the http method used in the request.

* Now you are going to add some code to the method. The code adds the product data, which the method receives from the request body, to the repository in case the productId, which is passed from the url to the method, not already exists.

Of course, it is also possible to send the product id together with the product data in the request body and process it from there, but for explanation purposes the first approach is chosen.

if (productRepository.addProduct (productId, product) {

final HttpHeaders headers = new HttpHeaders;

final URI uri =

builder.path(“drink/{id]”).buildAndExpand(productId).toUri();

headers.setLocation(uri);

return new ResponseEntity<>(product, headers, HttpStatus.CREATED);

} else {

return new ResponseEntity<>(product, headers, HttpStatus.CONFLICT);

}

**Give your shopping service an error handler**

* Again, open your shopping service from lab 2, if it is closed.
* If not already created, add a package with the name ‘exception’ to nl.sjop.service
* Add a ShoppingErrorHandler.java class to the nl.sjop.service.exception package and have it implement the org.springframework.web.client.ResponseErrorHandler interface.

You are going to register this error handler with the rest template further on. The error handler will be used to test response retrurned by the product service. In the example below, when the http status in the response not equals to 200 (OK) or 201 (CREATED), then the logic in the handleError method is invoked (by the springboot framework code).

* Add the implementation for the method hasError

@Override

public void hasError(ClientHttpResponse response) throws IOException {

return (response.getStatusCode() != HttpStatus.OK &&

response.getStatusCode() != HttpStatus.CREATED);

}

* Add the implementation for the method handlerError

@Override

public void handleError(ClientHttpResponse response) throws IOException {

throw new IOException(“Error in shopping service. Http status ” +

response.getStatusCode());

}

**Extend your shopping service**

Now you are going to extend the shopping service with a method for adding a product, which in turn will invoke the addDrink method in the product service via http POST.

* In the ShoppingController.java class, add a new method called addDrink

public static void addDrink(@PathVariable(“id”) String productId,

**@RequestBody** Product product) {

*[code to be implemented]*

}

* Annotate the method with a request mapping, so it it can be reached via http POST and url: [http://[servername]:[port]/shopping/drink/{id}](http://[servername]:[port]/shopping/drink/%7bid%7d)

(if you are not sure, you can have a look at the final code example, included at the end of this lab)

* The first thing for you to arrange in this method, is build an URI for invoking the productservice with. The builder takes care of the encoding for the characters in the specified url [http://[servername]:[port]/products/{productId}](http://[servername]:[port]/products/%7bproductId%7d) and replaces the placehoder productId with its actual value, as received in the path variable parameter named productId.

Copy the code that creates the URI from the method retrieveDrink, which you already added to ShoppingController.java file in lab 2.

* Next thing for you to take care of, is create a new instance of the RestTemplate and register the error handler you created in the preceding paragraph

RestTemplate restTemplate = new RestTemplate();

restTemplate.setErrorHandler(new ShoppingErrorHandler());

* Also prepare the http headers to send with the request to the product service

HttpHeaders headers = new HttpHeaders();

headers.setAccept(Arrays.asList(MediaType.APPLICATION\_JSON));

headers.setContentType(MediaType. APPLICATION\_JSON);

* Define an instance of HttpEntity, which will contain the data and headers to send in the rest call.

HttpEntity<Product> entity = new HttpEntity<>(product, headers);

* Invoke the product service by means of the exchange method in the RestTemplate class.

ResponseEntity<Product> result =

restTemplate.exchange(uri, HttpMethod.POST, entity, Product.class);

NOTE!

In the product service controller you already code the response for the addDrink: return new ResponsEntity<>(product, headers, HttpStatus.CREATED); or return new ResponseEntity<>(product, headers, HttpStatus.CONFLICT);

In this response entity, an instance of the class Product is returned. You have to make sure to match this with the generic specified in the ResponseEntity that is coded as the result for the exchange method

* For testing purposes, handle the result as follows:

if (result.getStatusCode() == HttpStatus.CREATED) {

System.out.println(“Product added”);

} else {

System.out.println(“Failure adding product with id: ”

+ productId);

}

* Test the shopping service:
  + Add a new product with a productId that does not exist. In the log see if the text “Product added” is printed
  + What is the response code for the addDrink method, and which one should you expect?
  + Repeat this test, with the same key. Why do you not see the failure message.

**Fixing the response for the addDrink method**

In the previous paragraph you defined the method addDrink as follows:

public static void addDrink(@PathVariable(“id”) String productId,

@RequestBodyProduct product) {

*[code to be implemented]*

}

So, the method doesn’t return a response entity, which it should!

Let’s fix this right now.

* Change the method declaration to:

public ResponseEntity<Product> addDrink((@PathVariable(“id”) String productId,

@RequestBodyProduct product) {

* Also add thefollowing line to the end of the method:

retrun result;

* Restart the shopping and product service and repeat the tests from the previous paragraph.
* Do you see http status 200 (first invoke) and http 500 (second invoke)?
* Why does it say http status 409 (= conflict) in the exception message and http status 500 (= internal server error) in the response?

**Put method**

The put method is used for updating data.

Can you implement this yourself?

**TODO**

**Add finalized code**