**Lab 4**

**Context and dependency injection**

In the preceding labs you ‘hard-wired’ the dependencies that exist in the shopping and product service. In this lab you will make changes to the dependencies, so that they are no longer ‘hard wired’.

**Refactoring the ProductService**

* Open your product service from lab 3, if it is closed.
* Open the file ProductRepistoryImpl.java and annotate the class as follows:

@Repository

Public class ProductRepositoryImpl {

[leave the code as it is]

}

* Extract an interface from the ProductRepositoryImpl.java class and name it ProductRepository. This can be done for you by tooling available in your IDE. In IntelliJ right click the ProductRepositoryImpl -> Refactor -> Extract -> Interface ->

Fill in the name for the interface and select the 4 methods.

package nl.groothandel.service.domain;

import java.util.Map

public interface ProductRepository {

void setProducts(Map<String,Product> products);

Map<String,Product> getProducts();

boolean addProduct(String id, Product product);

boolean updateProduct(String id, Product product);

}

* In the ProductsController.java file replace the hard wired ProductRepositoryImpl for just a property with the interface type:

~~ProductRepositoryImpl productRepository = new ProductRepositoryImpl();~~

ProductRepository productRepository;

Now, you can probably will see a message like the following “Could not autowire. No beans of ‘ProductRepository’ type found. You will fix this in the next bullet.

* In the ProductserviceApplication.java file replace the line with the @SpringBootApplication annotation with the following:

@SpringBootApplication(scanBasePackages = {“nl.groothandel.service”})

The scanBasePackages attribute is read by the Spring container, so it knows where to look/scan for beans (@Component, @Repository, @Service annotated Java classes) and to inject them. Sub folders from this base package are scanned as well, you do not need to specify them separately.

* Restart the product service and perform some tests to see thing are still working.

**Refactoring the ShoppingService**

* Open your shopping service from lab 3, if it is closed.
* In the file ShoppingController.java there are 4 lines for instantiating a rest template:

RestTemplate restTemplate = new RestTemplate();

These are hard wires, which you are going to replace.

* At the class level add a property referencing a RestTemplate instance, which will be autowired.

@Autowired

private RestTemplate restTemplate;

* Comment out the 4 lines mentioned above, that create a hard wired RestTemplate.

Do not remove them, as you will be using these below.

* If the method retrieveDrinks is defined as static, remove the static specifier.

Static methods cannot be autowired.

* In the ShoppingserviceApplication add a bean which exposes a RestTemplate:

@Bean

RestTemplate restTemplate() {

return new RestTemplateBuilder().build();

}

* In the ShoppingserviceApplication add ‘(scanBasePackages= {“nl.sjop.service”})’ to the @SpringBootApplication annotation (without the single quotes).
* Restart the shopping service and test if things still work.
* Another way to go, is to replace the 4 lines with the following:

RestTemplate restTemplate = restTemplateBuilder().build;

and replace the autowired RestTemplate with a RestTemplateBuilder :

@Autowired

private RestTemplateBuilder restTemplateBuilder;

. Also remove the bean that exposes the RestTemplate from the ShoppingserviceApplication.

* Restart and test the shopping service again.

**Add a 2nd ProductRepository to the ProductService**

Now you are going to add a second implementation of the ProductRepository interface to the product service.

* In the product service add a new class to the package nl.groothandel.service.domain and name it SodaProductRepositoryImpl.java
* Copy the content of the class ProductRepositoryImpl into the class you jyst created.
* Adjust the drinks that are available to something like this:

private Map<String, Product> products;

{

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

map.put(“COCACOLA”, new Product(“COCACOLA”,”Coca cola 1 liter”, “Cola”, “Coca Cola”, “1 liter”, 0, 1.59);

map.put(“PEPSICOLA”, new Product(“PEPSICOLA”, ”Pepsi cola”, “Cola”, “Pepsi”, “1 liter”, 0, 1.37);

this.products = map;

}

* Go to the ProductsController.java file.

As you can see it is invalid because there now exist 2 beans that conform to the ProductRepository interface.

* To fix this, you can annotate one of the 2 beans as primary. The Spring boot container will notice that this annotated bean is the preferred one, if more beans that qualify are available. Let’s do this for the SodaProductRepositoryImpl.java class :

@Repository

**@Primary**

public class SodaProductRepositoryImpl implements ProductRepository {

. . .

}

* Test the product service. Which drink products do you see?

**Profiles**

Normally you want more control over the choice for which bean to inject. This can be done at startup time using profiles. Let’s see how this works now.

* Add a new folder with the name configuration to the nl.groothandel.service package.
* Create e new LiquorConfiguration.java class into that folder
* Add the following annotations at the class level :

@Configuration

@Profile(“Liquor”)

public class LiquorConfiguration { . . . }

* Add the following bean declaration to the class :

@Bean

public ProductRepository productRepository() {

return new ProductRepositoryImpl();

}

As you will see, the spring container wil create an instance from the ProductRepositoryImpl.java class and expose it as a bean. It does so, if the profile Liquor is set at startup time, which you will do below.

* Create another configuration SodaConfiguration.java for creating a bean from the SodaRepositoryImpl.java class. Do this the same way as for the LiquorConfiguration above.
* Add a profile annotation to the ProductRepositoryImpl.java file:

@Repository

**@Profile(“liquor”)**

public class ProductRepositoryImpl . . .

* Add a profile annotation to the SodaProductRepositoryImpl.java file:

@Repository

**@Profile(“soda”)**

public class SodaProductRepositoryImpl . . .

* Set a profile to use at startup time. You can do this in the file application.properties as follows:

server.port=8081

**spring.profiles.active=soda**

* Test the product service. See that you get the soda products returned.
* Change the profile in the application.properties to the liquor value.

server.port=8081

**spring.profiles.active=liquor**

* Test the product service again. See that you get the liquor products returned.

**Spring 4.0 : Conditional**

With Spring 4.0 you can take a slightly different approach. Instead of using profiles, you can change over to conditionals for having the correct type of bean(s) created at startup time. Let’s change the product service again, to see how this works as well.

* First you are going to refactor the configuration classes:
  + rename the folder nl.groothandel.service.**configuration** to nl.groothandel.service.**condition**
  + rename the class LiquorConfiguration.java to LiquorCondition.java
  + rename the class SodaConfiguration.java to SodaCondition.java
  + Have both the class implement the org.springframework.context.annotation.Condition interface. This means that each class also implement the matches method. In this method you define the conditions that have to be matched for creating a specific bean.

public class SodaCondition implements Condition {

@Override

public boolean matches (ConditionContext context, AnnotatedTypedMetadata metdata) {

return “Soda”.equalsIgnoreCase(

context.getEnvironment().getProperty(“repository.name”));

}

public class LiquorCondition implements Condition {

@Override

public boolean matches (ConditionContext context, AnnotatedTypedMetadata metdata) {

return “Liquor”.equalsIgnoreCase(

context.getEnvironment().getProperty(“repository.name”));

}

* In the application.properties remove the line with the profile and replace it with:

repository.name=soda

* In the class ProductRepositoryImpl.java replace the profile(“liquor”) with:

@Conditional(value=LiquorCondition.class)

Note! At startup, the Spring container scans classes for bean annotions. Since the ProductRepositoryImpl is still annotated with @Repository it qualifies, but now only when it passes the matches test defines in the LiquorCondition.java class.

* Also change the SodaProductRepositoryImpl.java class

@Conditional(value=LiquorCondition.class)

* Test the product service again and also for the repository.name=liquor value.

**Difference between @Conditional and @Profile Annotations**

Both annotations are used to develop an “If-Then-Else” conditional checking. Where Spring Spring 3 profile checks are based on environment variables, Spring 4 Conditionals gives you a broader set of options to define your conditions with.