Exercise 10.1 – Spring REST

The Setup:

In this exercise we will create a RESTful customer web service. The shopping list RESTful web service example will be part of the code provided for this exercise to give you an executable example before building your own.

Open the **exercise10_1-client** project, and the **exercise10.1-service** project. Add the Spring context to both projects. Both projects also need the following dependency:

```
<dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.7.3</version>
</dependency>
```

The client project requires the following additional dependency:

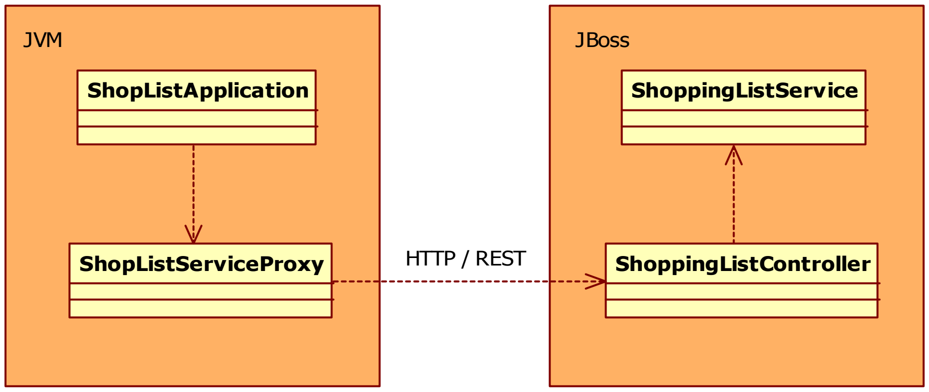
```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>4.0.2.RELEASE</version>
</dependency>
```

Since the service project is a web project it also requires the spring-webmvc dependency.

The Application:

The projects given include the complete shopping list RESTful web service example, and skeleton code for the customer RESTful web service for the exercise.

Run the service project (on the tomcat server).



The ShopListApplication makes calls to the ShopListServiceProxy as if it is the actual ShoppingListService object. If any objects are passed in the method call to the

ShopListServiceProxy uses the RestTemplate to marshall the message before sending it out on the HTTP request to the ShoppingListController. As the **MappingJackson2HttpMessageConverter** is on the classpath the RestTemplate uses it to marshall the message into JSON.

The ShoppingListController unmarshalls any objects, and passes the request on to the actual ShoppingListService. The ShoppingListService responds and any objects it returns to the ShoppingListController are marshalled again and sent back to the ShopListServiceProxy, which unmarshals them to provide the result to the ShopListApplication.

Running the **ShopListApplication.java** file (as java application) in the **Exercise10.1-Client** project should provide the following output:

```
3 Avocados     Organic or non-organic
3 Tomatoes     Prefer Organic

3 Avocados     Organic or non-organic
5 Tomatoes     Prefer Organic

5 Tomatoes     Prefer Organic
```

The Exercise:

Looking through the code, you will notice that we have already provided the **CustomerService** class and **ICustomerService** interface for this exercise in the services package of the **exercise10_1-service** project. We will explain this exercise step by step.

The Service:

To make the methods of the CustomerService class available for calling using Spring REST, we have to write a CustomerController in the **service** package of the **exercise10_1-service** project.

```
package service;

import org.springframework.stereotype.Controller;

@Controller
@RequestMapping(value="/customer")
public class CustomerController {

}
```

Annotate this class with the **@Controller** annotation to indicate that this is a SpringMVC controller.

Annotate the CustomerService with the @Service stereotype.

```
@Service
public class CustomerService implements ICustomerService {
```

Then, inject the CustomerService class into the CustomerController class.

```
	private CustomerService customerService;

	@Autowired
	public void setCustomerService(CustomerService customerService){
		this.customerService = customerService;
	}
```

Now, we have to write the methods of the CustomerController class. The CustomerController needs to implement the same methods as the CustomerService class.

Let's look at the different methods that the CustomerController needs to implement and its corresponding communication between the client and the server:

- **getCustomer(String customerNumber)**
  This method will be called when we receive a **GET /customer/{customerNumber}** request, and it should return the JSON representation of the corresponding customer.

- **getCustomers()**
  This method will be called when we receive a **GET /customers/** request, and it should return the JSON representation of the list of all customers.

- **addCustomer(Customer customer)**
  This method will be called when we receive a **POST /customer/{customerNumber}** request. Afterwards we want to redirect the caller back to the list of all customers, so that he can see the result of the add operation.

- **updateCustomer(String customerNumber)**
  This method will be called when we receive a **PUT /customer/{customerNumber}** request. Afterwards we want to redirect the caller back to the JSON representation of the updated customer, so that he can see the result of the update operation.

- **deleteCustomer(String customerNumber)**

This method will be called when we receive a **DELETE /customer/{customerNumber}** request. Afterwards we want to redirect the caller back to the list of all customers, so that he can see the result of the delete operation.

There are 2 methods that return objects directly: getCustomer() and getCustomers().

This is a general web development principle; no method other than GET should ever return data. POST, PUT, and DELETE should always return a redirect to a GET that then provides the data. This to prevent duplicate submissions due to browser refreshes.

Our method stubs therefore become:

```
@Controller
@RequestMapping(value="/client")

public class CustomerController {
  private CustomerService customerService;
  public void setCustomerService(CustomerService customerService){
    this.customerService = customerService;
  }

  public void addCustomer(@RequestBody Customer customer) {}
  public void deleteCustomer(@PathVariable("customerNumber") String
          customerNumber) {}
  public Customer getCustomer(@PathVariable("customerNumber") String
          customerNumber) {}
  public Collection<Customer> getCustomers() {}
  public void updateCustomer(@RequestBody Customer customer) {}
}
```

Although the method parameters on the controller are identical to the method parameters on the CustomerService class, they have to be mapped to a specific part of the incoming HTTP request using the **@PathVariable** and **@RequestBody** annotations. When using the **@PathVariable** annotation, be sure to specify the name of the path variable to which you are mapping.

Use **@RequestBody** when you receive an **object** as method parameter. In this case, both the methods addCustomer() and updateCustomer() receive the Customer object as a parameter.

Use **@PathVariable** when you receive a **variable** as method parameter. In this case, both the methods deleteCustomer() and getCustomer() receive the variable customerNumber as a parameter.

```
@Controller
public class CustomerController {
  private CustomerService customerService;
  public void setCustomerService(CustomerService customerService){
    this.customerService = customerService;
  }

  @RequestMapping(value = "/customer", method = RequestMethod.POST)
  @ResponseStatus(HttpStatus.CREATED)
  public void addCustomer(@RequestBody Customer customer) {}

  @RequestMapping(value = "/customer/{customerNumber}", method =
                  RequestMethod.DELETE)
  @ResponseStatus(HttpStatus.OK)
  public RedirectView deleteCustomer(@PathVariable("customerNumber") String
         customerNumber) {}

  @RequestMapping(value = "/customer/{customerNumber}", method =
                  RequestMethod.GET)
  public @ResponseBody Customer getCustomer(@PathVariable("customerNumber")
         String customerNumber) {}

  @RequestMapping(value = "/customers", method = RequestMethod.GET)
  public @ResponseBody Collection<Customer> getCustomers() {}

  @RequestMapping(value = "/customer/{customerNumber}", method =
                  RequestMethod.PUT)
  @ResponseStatus(HttpStatus.OK)
  public void updateCustomer(@RequestBody Customer customer) {}
}
```

Add the **@RequestMappings** to the each method, and specifying the URL and the HTTP RequestMethod to which this method should be mapped.

Next, call the corresponding methods of CusomerService to the CustomerController class.

When ready, run your application on the server. Remember that the page shown is an HTML just for your information. Normally it would show a 404 error because there is no resource available at the requested URL.

The Client:

Now, we have to implement the client application that will call our REST Customer application.

Create the **CustomerServiceProxy** class that implements **ICustomerService** in the **service** package in the Client project.

```
package service;

import generated.customer.Customer;
import generated.customer.Customers;
import java.util.Collection;

public class CustomerServiceProxy implements ICustomerService{

      public void addCustomer(Customer customer) {

      }

      public void deleteCustomer(String customerNumber) {

      }

      public Customer getCustomer(String customerNumber) {

      }

      public Collection<Customer> getCustomers() {

      }

      public void updateCustomer(Customer customer) {

      }
}
```

Then, insert the CustomerServiceProxy in the Spring configuration file and inject the RestTemplate into the CustomerServiceProxy.

```
<bean id="customerServiceProxy" class="service.CustomerServiceProxy">
      <property name="restTemplate" ref="restTemplate"/>
</bean>
```

```
public class CustomerServiceProxy implements ICustomerService{

      private RestTemplate restTemplate;
      public void setRestTemplate(RestTemplate restTemplate) {
            this.restTemplate = restTemplate;
```

```
        }
        …
```

Then, add 3 private static String attributes that contain the URL to the REST Customer service that will be running in Jboss.

```
public class CustomerServiceProxy implements ICustomerService{
      private static final String customersURL = 
"http://localhost:8080/exercise10_1-service/rest/customers";
      private static final String newCustomersURL = 
"http://localhost:8080/exercise10_1-service/rest/customers";
      private static final String customerURL = 
"http://localhost:8080/exercise10.1-service/rest/customer/{customerNumber}";
      …
```

Lastly, we have to implement the methods of the CustomerServiceProxy using the RestTemplate.

```
  private RestTemplate restTemplate;
  public void setRestTemplate(RestTemplate restTemplate) {
    this.restTemplate = restTemplate;
  }

  public void addCustomer(Customer customer) {
    restTemplate.postForLocation(newCustomersURL, customer);
  }

  public void deleteCustomer(String customerNumber) {
    restTemplate.delete(customerURL, customerNumber);
  }

  public Customer getCustomer(String customerNumber) {
    return restTemplate.getForObject(customerURL, Customer.class, 
           customerNumber);
  }

  public Collection<Customer> getCustomers() {
    Customers customers = restTemplate.getForObject(customersURL, 
                          Customers.class);
    return customers.getCustomer();
  }

  public void updateCustomer(Customer customer) {
    restTemplate.put(customerURL, customer, customer.getCustomerNumber());
  }
}
```

Update the **CustomerApplication** class in the client project to create 3 customers, and print the resulting list of customers. Then, update a customer and print list. Lastly, delete a customer and again print the results.

Exercise 10.2 – Spring HTTP Invoker Calculator

The Setup:

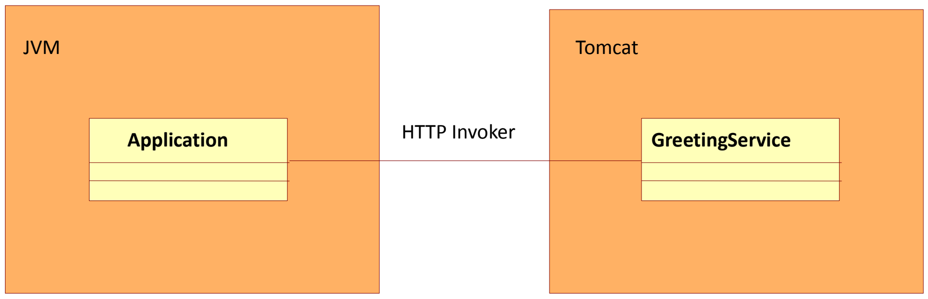
In this exercise, we will implement a calculator using the Spring HTTP invoker. We provide the code for the working greeting example as reference material.

Start by opening the **exercise10_2-service** and **exercise10_2-client** projects. Both projects depend on **spring-web**, the service project also depends on **spring-webmvc**.

The Application:

The Application uses the remote GreetingService as if it's a local object through the generated greetingHttpInvokerProxy.

The big difference between the other web service technologies and the HTTP invoker is that we do not have to write a proxy or an endpoint. Both are automatically generated from our Spring configuration.



To run the example code, start the **exercise25.2-Service** project on the tomcat server. Once the service is running, if you ran it on Eclipse the browser will show a 404 error because there is no resource available at http://localhost:8080/exercise10_2-service/. Then go to the client project and run the **Application** class in the project should produce the following output:

```
Receiving result: Hello John Doe
```

The Exercise:

Study the code, once you are comfortable with how the HttpInvoker works, create a stateful calculator. In other words, the Service side should maintain the current 'state' as a int value, and an 'calc' method that takes an operator (char) and value(int) that change this state and also returns the updated value of 'state'. (For thread safety it would be good to make it synchronized).

Once you have completed your calculator, add stopwatch functionality to the client to output the duration of each remote call.