Enterprise Java with Spring
Spring REST API
Lab 4

1 Lab setup

Make sure you have the following items installed

- Latest LTS JDK version (at this point: either JDK 21)
- Spring Tool Suite (STS) or IntelliJ IDEA
- Latest version of Maven (at this point: Maven 3.9.9)
- A free account at Postman and installed the Postman app
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`. The `changes` subfolder just holds the source code files for the lab, while the `final` subfolder holds the complete Eclipse project starting from its project root folder. We will use the code from the `changes` subfolder to build up our applications from scratch and you can always fall back on the complete Eclipse project if you encounter any errors while building up the application.

2 Creating a RestTemplate client

The source code for this lab is found in `Basic-Rest-Template/changes` folder.

Start up STS. Ensure you are in the Java EE perspective.

Go to File -> New -> Other -> Spring Boot -> Spring Starter Project. Complete it with the following details:

Name: `BasicRestTemplate`
Group: `com.workshop.rest`
Artifact: `BasicRestTemplate`
Version: `0.0.1-SNAPSHOT`
Description: `Simple Rest Template client to consume REST API`
Package: `com.workshop.rest`

Click Next.

Add the following dependencies:
```
Web -> Spring Web
Developer Tools -> Spring Boot Dev Tools
```

In `src/main/resources`, place the files

```
application.properties
```

In the package `com.workshop.rest` in `src/main/java`, place the files

```
MainConfig
MyRestService
MyRunner
```

The RestTemplate class is a synchronous client used for calling and consuming RESTful web services. It allows Spring applications to make HTTP requests (GET, POST, PUT, DELETE, etc.) to APIs, interpret the responses, and convert them into Java objects. It simplifies the process of sending HTTP requests and handling HTTP responses in Java applications, abstracting much of the boilerplate code required for working with APIs.

There are several ways to instantiate a RestTemplate object: we demonstrate two simple approaches in MainConfig and mark the approach we want to use with @Primary. You can switch between these 2 approaches by placing the @Primary annotation on the particular bean method whose implementation you want to use in order to instantiate a RestTemplate object

This object is injected using @Autowired into MyRestService that functions as a sort of generic DAO which is subsequently accessed in the MyRunner class whose run method is immediately executed when the application boots.

We have configured this application to run at the console without a web server (through a property in `application.properties`) so as not to interfere with running other REST API services in the embedded Tomcat server at default port of 8080.

The base URL for the REST API calls is also defined in `application.properties`, so make sure to change it here if you wish to make calls to a different REST service.

Start up the app in the usual manner.

Right click on the project entry in the Project explorer pane and then select Run As -> Spring Boot App, or right click on the project entry in Spring Boot dashboard (Window -> Show View -> Other -> look in the Other folder -> Boot Dashboard) and select Start / Restart.

Verify that log output appears from the `run` method of MyRunner

3 RestTemplate methods

RestTemplate class provides a wide range of methods for sending standard REST API HTTP requests to a REST service and processing the returned response.

The most common forms are:

- `xxxxForObject` – operates on a resource representation where the response (if any is provided) is deserialized into a given class
- `xxxForEntity` – operates on a resource representation where the response (if any is provided) is deserialized into a ResponseEntity. This allows access to the HTTP headers of the response.
- `exchange` – allows the construction of a RequestEntity and obtain the response as a ResponseEntity.

3.1 Using xxx and xxxForObject method calls

In the package `com.workshop.rest` in `src/main/java`, place the files

```
Employee
Resume
```

In the package `com.workshop.rest` in `src/main/java`, make the following changes

```
MyRestService-v2
MyRunner-v2
```

Here we use `xxxForObject` method calls to deserialize the body of the response into an object for the GET and POST methods. For the PUT and DELETE methods, we use the simplest form of method call `xxx` as we are not expecting any response or we do not wish to process the response.

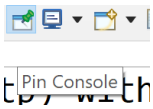
We will use the `RestMethodsResponse` app that we constructed in Lab 2 as the REST API service to test our RestTemplate client against.

Start up the `RestMethodsResponse` app in the usual manner from the Boot dashboard.

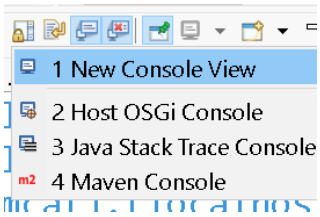
In order to see the Console output from these two applications, you will have to:

- pin the current console view showing output from `RestMethodsResponse`
- open a new separate Console view
- ensure that the new Console view does not switch to a different application when standard output changes
- Run `BasicRestTemplate` in the new Console view

In the current Console view, select Pin Console from the icon at the upper right hand corner of the Console View



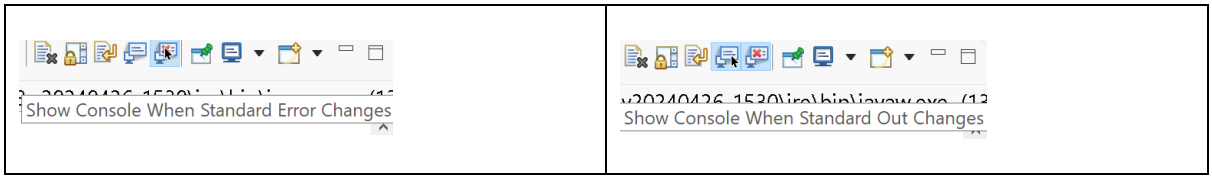
Then, select New Console View from the drop-down list of the icon at the upper right hand corner of the Console View.



Switch over to this new Console view. You will notice it is currently identical to the older one. Make sure both the following options

- Show Console When Standard Out changes
- Show Console When Standard Error changes

are deselected from the icons at the upper right hand corner of the Console View



With this new console view active, select `BasicRestTemplate` from the Boot Dashboard and run it. You should see the log output from `BasicRestTemplate` appear in this new console view, while the old console view shows the latest log output from `RestMethodsResponse`

Verify that the GET, POST, PUT and DELETE calls produce the correct results in the console views for these 2 applications. You can open up the EmployeeController from `RestMethodsResponse` to help in your verification.

You can choose to use the Eclipse TCP/IP monitor to monitor HTTP traffic between the `RestMethodsResponse` REST service and `BasicRestTemplate` client, if you wish. If you are using the Eclipse TCP/IP monitor, make sure to change the base URL for `myrest.url` in `application.properties` as appropriate. For e.g. if your monitor is listening on port 9090 and redirecting to port 8080, then change this to:

```
myrest.url=http://localhost:9090/api
```

3.2 Using xxxForEntity method calls

In the package `com.workshop.rest` in `src/main/java`, make the following changes

`MyRestService-v3`
`MyRunner-v3`

The `xxxForEntity` method is useful to obtain HTTP header as well as status information from a returned response via a ResponseEntity.

Ensure that the `RestMethodsResponse` app is still active and running.
Restart the app in the usual manner.

Verify that the 2 GET calls to the different URLs produce the correct results in the console views for these 2 applications. You can open up the HeaderController from `RestMethodsResponse` to help in your verification.

3.3 Using exchange method calls

In the package `com.workshop.rest` in `src/main/java`, make the following changes

`MyRestService-v4`
`MyRunner-v4`

The `exchange` method is useful to place HTTP headers into the outgoing request as well as obtain HTTP header and status information from a returned response.

Restart the app. Ensure that the `RestMethodsResponse` app is still active and running.

Verify that the GET call to the URL produces the correct results in the console views for these 2 applications. You can open up the HeaderController from `RestMethodsResponse` to help in your verification.

3.4 Sending path and query parameters

In the package `com.workshop.rest` in `src/main/java`, make the following changes

`MyRestService-v5`
`MyRunner-v5`

Although it is possible to include path and query parameters in the URL that we are invoking by hardcoding them directly into the URL string itself, we can also build it up gradually if so required

Restart the app. Ensure that the `RestMethodsResponse` app is still active and running.

Verify that the GET call to the URL produces the correct results in the console views for these 2 applications. You can open up the ParamsController from `RestMethodsResponse` to help in your verification.

Stop all running apps.

4 Interacting with public APIs

We can also use our RestTemplate client to interact with public REST APIs (rather than a local REST API service), which will be the most common use case.

The primary issue with interacting with public REST APIs is to determine the format of the class to deserialize incoming JSON content from HTTP responses into, and conversely, the format of the class to serialize into JSON content for outgoing HTTP requests.

We may also need to insert API keys for authentication / authorization purposes into either specific headers or specific path portions of the outgoing HTTP requests.

We can determine the structure of the classes by examining the JSON content beforehand through prior interaction using an external REST client such as Postman.

4.1 Interacting with jsonplaceholder

For e.g. making a GET request to:

https://jsonplaceholder.typicode.com/posts

returns content similar to the following:

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi
optio reprehenderit",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur
expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum
rerum est autem sunt rem eveniet architecto"
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit
dolor beatae ea dolores neque\nfugiat blanditiis voluptate porro vel
nihil molestiae ut reiciendis\nqui aperiam non debitis possimus qui
neque nisi nulla"
  },
….
….
```

We can then create a Java class with field names and types that match the key / value pairs in this JSON content.

In the package `com.workshop.rest` in `src/main/java`, place the files

```
Post
Comment
```

In the package `com.workshop.rest` in `src/main/java`, make the following changes

```
MyRestService-v6
MyRunner-v6
```

In `src/main/resources`, make the following changes

```
application.properties-v6
```

Restart the app.
Verify the log output matches the response content that would be received via Postman or some other REST API client.

4.2 Interacting with exchangerate API

The exchange rate REST API is a popular free API service that provides latest up to date exchange rates for popular global currencies and is used by main applications in global companies.

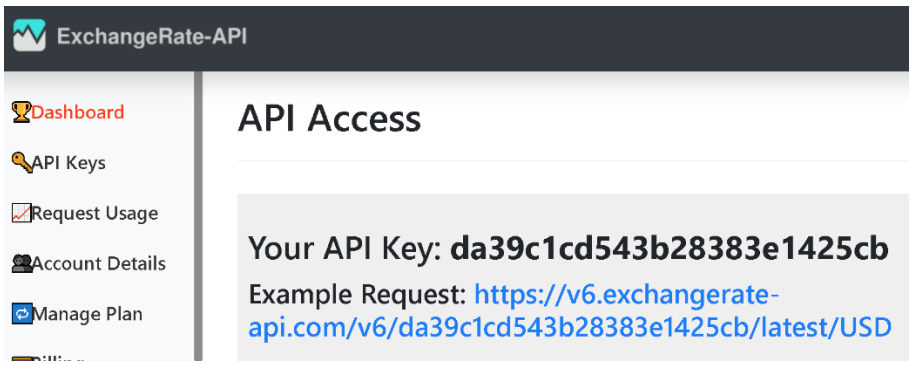
It requires an API key to be inserted into the path portion of the request URL:

The format of the URL is as follows:

```
GET     https://v6.exchangerate-api.com/v6/YOUR-API-KEY/latest/base-currency
```

To get the key, you will first need to create a free account and then sign in.

After performing this, you should then be able to get an API key access similar to the one shown below on the main dashboard:



If you attempt to make a request to the sample URL (which includes your API access key) either using your browser or Postman, you will notice that the JSON content is extremely long for the last key `conversion_rates`, as this shows the conversion of that specified currency against the top global currencies.

Thus, programmatically speaking, it is not feasible to create a class to mirror this content in order to deserialize it, as we have being doing so far. Instead, we can obtain the response as a String, and parse it using the JSON library that is used in Spring Web: Jackson databind.

In the package `com.workshop.rest` in `src/main/java`, make the following changes

```
MyRestService-v7
MyRunner-v7
```

In `src/main/resources`, make the following changes

```
application.properties-v7
```

It is conventional to specify the API key as a value in `application.properties` that we can subsequently read into MyRestService via the @Value annotation.

In MyRestService, we parse the String received as a response as a JSON object and drill into the `conversion_rates` node. From there, we can extract the relevant values for the keys we want representing the currencies to convert into.

Restart the app.
Verify the log output matches the response content that would be received via Postman or some other REST API client.

5 Working with fully implemented REST API service

Here, we will work with the DevSpringRest app we created in a previous lab.

In the package `com.workshop.rest` in `src/main/java`, place the file:

```
Developer
```

In the package `com.workshop.rest` in `src/main/java`, make the following changes

```
MyRestService-v8
MyRunner-v8
```

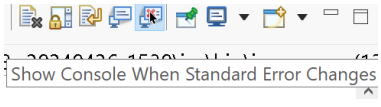
In `src/main/resources`, make the following changes

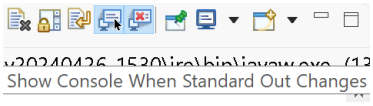
```
application.properties-v8
```

Make sure both the following options

- Show Console When Standard Out changes
- Show Console When Standard Error changes

are deselected from the icons at the upper right hand corner of the Console View





Start DevSpringRest from the Boot dashboard in one console view

Restart BasicRestTemplate in another console view.

Press enter to step through the actions in MyRunner one at a time, and verify the log output in both console views matches the expected results from these actions.

Notice that the `postForLocation` call in `addDeveloper` in MyRestService returns a URI. This is because by default, the logic that services a REST POST call should provide the URI to retrieve the newly created source in the Location header of the response. The `postForLocation` extract this URL and returns it as a URI object.

Keep in mind that if you wish to rerun BasicRestTemplate, you also need to restart DevRestApp as well because the changes made to the in-memory list of developers in DevRestApp from the first run of BasicRestTemplate will cause errors when you attempt to run it again.

6 Using Lombok

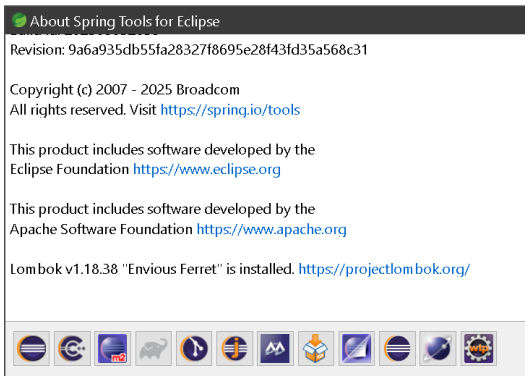
Project Lombok is used to minimize boiler-plate code that frequently appear in classes such as getters, setters, constructors, etc. It accomplishes this by plugging into the build process and auto-generating Java bytecode into the `.class` files based on Lombok-specific annotations that we introduce in our code.

Close STS.

Download the Lombok main JAR file

Double click on the JAR file, and select your STS for your installation.

Start up STS again. From the main menu, select Help -> About Spring Tool Suite 4. You should have a statement at the end confirming the installation of Lombok.



Add the following dependency snippet to your `pom.xml` of `BasicRestTemplate`. Make sure that the version matches that shown in the dialog box above.

```
		<dependency>
			<groupId>org.projectlombok</groupId>
			<artifactId>lombok</artifactId>
			<version>1.18.38</version>
		</dependency>
```

Save the POM and do Maven -> Update Project, and follow up with a project refresh.

In the package `com.workshop.rest` in `src/main/java`, make the following changes

`Developer-v9`
`MyRunner-v9`

Notice that all the constructors, toString, and individual field setter and getter methods in Developer here have been replaced with their related Lombok annotation.
In MyRunner, the new Lombok @Slf4j annotation in the background creates the logger with the following statement:

```
private static final org.slf4j.Logger log =
    org.slf4j.LoggerFactory.getLogger(MyRunner.class);
```

Start DevSpringRest from the Boot dashboard in one console view

Restart BasicRestTemplate in another console view.

Press enter to step through the actions in MyRunner one at a time, and verify the log output in both console views matches the expected results from these actions as we did before previously, to show that functionality remains identical with the introduction of Lombok to minimize boiler-plate code

Close both apps.

In the package `com.workshop.rest` in `src/main/java`, make the following changes

`Developer-v10`

Here we show a way to further simply the annotation for Developer.

@Data generates all the boilerplate that is normally associated with simple POJOs (Plain Old Java Objects) and beans:

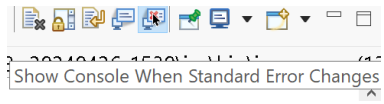
- getter methods for all fields,
- setter methods for all non-final fields,
- appropriate toString(),
- appropriate equals()
- hashCode() implementations that involve the fields of the class
- constructor that initializes all final fields,

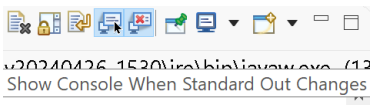
- constructor that initializes all non-final fields with no initializer that have been marked with @NonNull

Make sure both the following options

- Show Console When Standard Out changes
- Show Console When Standard Error changes

are deselected from the icons at the upper right hand corner of the Console View





Stop and restart DevSpringRest from the Boot dashboard in one console view

Restart BasicRestTemplate in another console view.

Press enter to step through the actions in MyRunner one at a time, and verify the log output in both console views matches the expected results from these actions as we did before previously, to show that functionality remains identical with the introduction of Lombok to minimize boiler-plate code

Close both apps.