Spring REST Workshop
Lab 2

1   Lab setup

Make sure you have the following items installed

- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Spring Tool Suite (STS).
- Latest version of Maven
- A free account at Postman and installed the Postman app
- A suitable text editor (Notepad ++)
- A utility to extract zip files

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   REST HTTP methods and response types

The main folder for this lab is `Rest-Methods`

Start up STS. Switch to the Java perspective.

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

Name: `RestMethodsResponse`
Group: `com.workshop.rest`
Artifact: `RestMethodsResponse`
Version: `0.0.1-SNAPSHOT`
Description: `Demo Spring Rest methods and response types`
Package: `com.workshop.rest`

Add the following dependencies:
`Web -> Spring Web`

Replace `pom.xml` for the project with the `pom.xml` from `changes`

We include the following dependency in `pom.xml` (in addition to the standard Spring Boot starters) in order to be able to return XML content. Note that this dependency already exists in the dependency hierarchy

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
  <version>2.12.4</version>
</dependency>
```

In `src/main/resources`, place the file `logback-spring.xml`

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

```
Employee
Resume
EmployeeController
RestMethodsResponseApplication
```

EmployeeController uses @Autowired to initialize its two fields `myEmployee` and `myResume`; the initialization code for this is available in RestMethodsResponseApplication via the @Bean factory methods. This is possible because the @SpringBootApplication annotation incorporates 3 other annotations implicitly (@Configuration, @EnableAutoConfiguration and @ComponentScan).

Notice that we have path mappings for all common REST HTTP methods (Get, Post, Put, Delete) with the exception of Patch (which we will look at in a later lab session).

The Mime (or media) type for the response can be specified along with the path mapping as a string (`"application/json"`, `"application/xml"`, etc). If none is specified, the default is JSON (`"application/json"`). This can be specified explicitly if desired using the `produces` attribute and the corresponding String value. Alternatively, we can use the MediaType constants provided by Spring framework:

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/MediaType.html

Similarly, we can also specify the type of data in the HTTP Request uses the `consumes` attribute.

It is possible to specify more than one type (usually it will be JSON and XML) for the return or request type in the body. Different types can be mapped to the same endpoints but handled by different mapping methods. Alternatively, you can have a single method being mapped to different types for the same endpoint.

2.1   Testing GET, POST, PUT and DELETE

We will test all the REST methods that are mapped in EmployeeController using Postman.

Create a new collection to save your requests: `REST lab requests`

Start up the app in the usual manner.
You can also use the Eclipse TCP/IP monitor or Rawcap/Wireshark to help out in debugging any issues that arise.
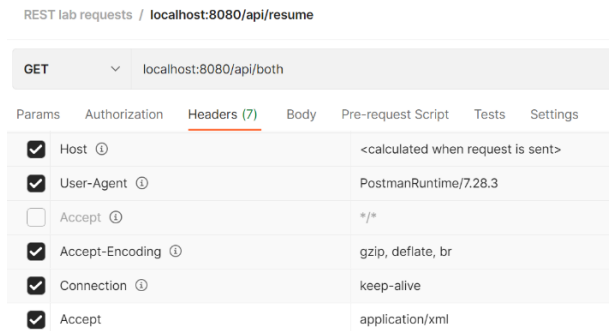
Create new GET requests to:

```
localhost:8080/api/resume
localhost:8080/api/employee
```

Verify the JSON content is returned and the appropriate log messages corresponding to the mapped methods appear in the console view.

Create a new GET request to:

```
localhost:8080/api/both
```

Set up the Accept header with a new value of `application/xml` and unselect the preset header with the value of */*

Send the request and verify the content returned is XML and the appropriate log messages corresponding to the mapped methods appear in the console view.

Modify the Accept header now to `application/json`.

Send the request and verify the content returned is JSON and the appropriate log messages corresponding to the mapped methods appear in the console view.

Create a new POST request to:

`localhost:8080/api/resume`

Set the body to `raw` with XML format and enter this as its content:

```
<Resume>
    <university>Oxford</university>
    <spokenLanguages>
        <spokenLanguages>German</spokenLanguages>
        <spokenLanguages>Malay</spokenLanguages>
    </spokenLanguages>
    <skillSets>Fantastic project manager</skillSets>
    <yearsExperience>3</yearsExperience>
    <cgpa>3.25</cgpa>
</Resume>
```

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

Set the body to `raw` with JSON format and enter this as its new content:

```
{
    "university": " IIT",
    "spokenLanguages": [
        "Hindi",
        "Tamil"
    ],
    "skillSets": "Cool CEO",
    "yearsExperience": 15,
    "cgpa": 3.99
```

```
}
```

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

Create a new POST request to:

`localhost:8080/api/employee`

Set the body to `raw` with JSON format and enter this as its content:

```
{
    "name": "Spiderman",
    "age": 22,
    "resume": {
        "university": "Marvel Universe",
        "spokenLanguages": [
            "English",
            "Chinese",
            "Russian"
        ],
        "skillSets": "Great webslinger",
        "yearsExperience": 2,
        "cgpa": 4.0
    },
    "married": false
}
```

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

Create a new POST request to:

`localhost:8080/api/vals`

Set the body to `x-www-form-urlencoded` and enter some random key-value pairs:

	KEY	VALUE
☑	name	spiderman
☑	age	33

none   form-data   ● x-www-form-urlencoded   raw

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

The type `x-www-form-urlencoded` is used for HTML form data submitted via a POST request. Notice that we now need to use @RequestParam and a Map structure to extract the data, rather than deserializing into a specific class structure using @RequestBody.

Create a new PUT request to:

`localhost:8080/api/resume`

Set the body to `raw` with JSON format and enter this as its content:

```
{
    "university": " IIT",
    "spokenLanguages": [
        "Hindi",
        "Tamil"
    ],
    "skillSets": "Cool CEO",
    "yearsExperience": 15,
    "cgpa": 3.99
}
```

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view showing the content received and deserialized correctly.

Create a new DELETE request to:

`localhost:8080/api/resume`

Select `None` for the body content.

Send the request and verify that the log messages corresponding to the mapped methods appear in the console view.

Notice that for all cases so far, successful invocation of the mapped method in the @RestController class returns a status 200 OK by default.

2.2   Common client-side error messages

Lets examine some of the messages that are returned by default by the Spring framework due to client-side errors. The message typically includes a timestamp, a HTTP status code and error message and the path portion of the URL.

The most common client-side HTTP error messages are:
- 404 Not Found
- 405 Method Not Allowed
- 400 Bad Request
- 415 Unsupported Media Type

The Spring MVC exceptions that correspond to this errors are listed in:
https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-ann-rest-spring-mvc-exceptions

Try to send a GET, POST, PUT or DELETE request to a URL with no matching mapping, for e.g:

`localhost:8080/api/asdfasdf`

Verify a 404 Not Found error message is received in return. Note that the log output on the service does not show anything as this error is intercepted and handled by the embedded Tomcat server itself.

Try to send a PATCH request to this URL:

`localhost:8080/api/resume`

Verify a 405 Method not allowed error message is received in return. Check the log output on the service to see the warning message displayed.

Try to send a POST request to:

`localhost:8080/api/resume`

with the raw JSON content of

```
{
    "city": "Metropolis",
    "population": 10000
}
```

Although a 200 OK Status is received, notice that the log messages on the server side show that deserialization results in an object with null fields.

Resend the POST request to:

`localhost:8080/api/resume`
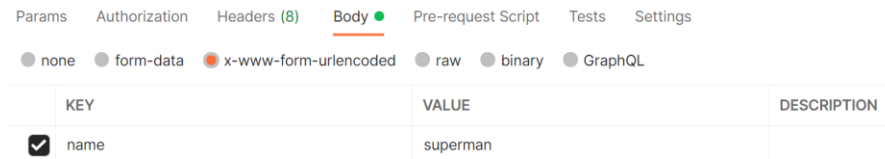
with the malformed raw JSON content of

```
{
    "city : "Metropolis",
    "population": 10000
}
```

Verify a 400 Bad Request error message is received in return. Check the log output on the service to see the warning message displayed.

Resend the POST request to:

`localhost:8080/api/resume`

in the `x-www-form-urlencoded` format with a dummy key and value pair:

Verify a 415 Unsupported Media Type error message is received in return. Check the log output on the service to see the warning message displayed.

3   Accessing Path and Query parameters

In `src/main/java`, place the file

`ParamsController`

Restart the app

Notice that we can have multiple @RestController classes together as long as they provide unique request mappings for their methods and they are in same package (or subpackage) that the @SpringBootApplication class is in.
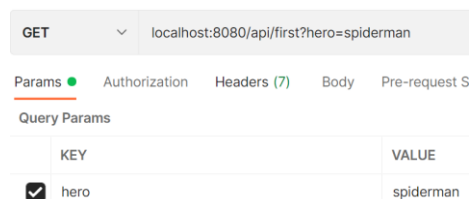
In this session, we will demonstrate the variety of ways to access path and query parameters. This is demonstrated using only @GetMapping methods, but they are equally applicable to all standard REST HTTP method mappings.

3.1   Accessing query parameters via @QueryParam

Make a GET request to:

localhost:8080/api/first?hero=spiderman

You can either type the query string directly or set it up via the Params tab in Postman.

Verify that the value is displayed correctly in the log output on the server side

Make a GET request to:

`localhost:8080/api/second?person=superman&age=33`

Verify that the values are displayed correctly in the log output on the server side

Make a GET request to:

`localhost:8080/api/first`

Notice that a 400 Bad Request is returned as the mapping method expects a compulsory query parameter with the key hero.

Make a GET request to:

`localhost:8080/api/third?hero=spiderman`

Verify that the value is displayed correctly in the log output on the server side

Repeat the GET request without the parameter:

`localhost:8080/api/third`

Verify that the correct statement appears in the log output on the server side

Make a GET request to:

`localhost:8080/api/fourth?hero=spiderman`

Verify that the value is displayed correctly in the log output on the server side

Make a GET request to:

`localhost:8080/api/fourth`

Verify that the default parameter value is displayed correctly in the log output on the server side

Make a GET request to:

`localhost:8080/api/fifth?person=superman&age=33&job=journalist`

Verify that the 3 key-value pairs are displayed correctly in the log output on the server side

Make a GET request to:

`localhost:8080/api/sixth?heroes=ironman,black widow,thor`

Verify that the 3 values for `heroes` are displayed correctly in the log output on the server side

3.2   Accessing path parameters via @PathVariable

Make GET requests to:

```
localhost:8080/api/seventh/ironman
localhost:8080/api/seventh/superman
```

Verify that the values are displayed properly at the server side

Make GET requests to:

```
localhost:8080/api/eighth/ironman/22
localhost:8080/api/eighth/black-widow/35
```

Verify that the values are displayed properly at the server side

Make a GET request to:

```
localhost:8080/api/ninth/Jane/manager/female
```

Verify that the values are displayed properly at the server side

Make a GET request to:

```
localhost:8080/api/ninth/Jane/manager
```

Notice a 404 Not Found error is returned.

Make a GET request to:

```
localhost:8080/api/tenth/Jane
```

Verify that the value is displayed properly at the server side

Make a GET request to:

```
localhost:8080/api/tenth
```

Verify that the correct statement appears in the log output on the server side

There are no default values for @PathVariable, unlike @RequestParam. Default values will have to be explicitly included in the logic of the path processing code.

4   Setting HTTP response status codes

In `src/main/java`, place the file

`ResponseController`

Restart the app

4.1　Using @ResponseStatus and ResponseEntity

So far, all of the successfully executed request mapping methods return an implicit 200 OK status. We can also choose to provide other HTTP status codes in the response. There are two ways to do this (using @ResponseStatus and the ResponseEntity object). The status codes available are provided in:

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpStatus.html

Note that this is only a subset of the full range of HTTP status codes officially available:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Make a GET request to:

```
localhost:8080/api/status-one
```

and verify a 202 Accepted status code is returned. Experiment with changing to other status codes and repeat the GET request.

Make a GET request to:

```
localhost:8080/api/status-two
```

and verify a 301 Moved Permanently status code is returned. Experiment with changing to other status codes and repeat the GET request.

5　Accessing HTTP Headers

In `src/main/java`, place the file

```
HeaderController
```

Restart the app

5.1　Retrieving request headers with @RequestHeader

We demonstrate setting request headers here using only @GetMapping methods, but it works exactly the same for any of the other standard REST HTTP methods.

Make a GET request to:

```
localhost:8080/api/header-first
```

and check the Host and User-Agent header values in the Headers tab before sending the request.

Verify that these header values appear in the log output on the server side.

Make a GET request to:

`localhost:8080/api/header-second`

Verify that the appropriate header values appear in the log output on the server side.

Make a GET request to:

`localhost:8080/api/header-third`

Verify that all header values appear in the log output on the server side.

Make a GET request to:

`localhost:8080/api/header-fourth`

Verify that the appropriate statement appears in the log output on the server side

Add a header for `hero` with an appropriate value in the Headers view and repeat the request.

Verify that the header value appears in the log output on the server side

Make a GET request to:

`localhost:8080/api/header-fifth`

Verify that the default value for the `job` header appears in the log output on the server side

Add a custom value for the `job` header and repeat the request.

Verify that the set value for the `job` header now appears in the log output on the server side

5.2 Adding response headers with HttpServletResponse and ResponseEntity

In `src/main/java`, place the file

`AddResponseHeaderFilter`

and make the change:

`RestMethodsResponseApplication-v2`

Restart the app.

Make a GET request to:

`localhost:8080/api/header-sixth`

Check in the response view in Postman for the `hero` header.

Make a GET request to:

`localhost:8080/api/header-seventh`

Check in the response view in Postman for the `job` header.

A @PostMapping method will typically persist/store the JSON content sent to it from the client. To help the client fetch this stored content in the future, it will also typically return a URL which can be used with a future GET request for this purpose. This URL is typically provided as the value of the `Location` header in the response from the @PostMapping method and the response is returned with a status code of 201 Created. Here, we pretend that a new resource has been created which is accessible at the current path URL with the resource ID appended to its end and return this in the `Location` header

Make a POST request to:

`localhost:8080/api/header-eighth`

Check in the response view in Postman for the `Location` header.

We can add filters to process incoming requests or responses, so that certain actions are performed to specific requests before they are processed by the server or to specific responses before they are returned to the client. This helps us to avoid repeating code to add response headers to all our relevant @RestController methods. This can be done by implementing a Filter which is then registered as a FilterRegistrationBean in the @SpringBootApplication class.

Make GET requests to:

```
localhost:8080/api/filter-header/first
localhost:8080/api/filter-header/second
```

Check in the response view in Postman for the `city` header.

6   Handling exceptions in REST

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

```
SampleController
ResourceNotFoundException
AnotherCustomException
```

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

```
application.properties
```

So far, we have seen a variety of client side errors resulting in a variety of error messages returned back from the server, such as:
- 404 Not Found
- 405 Method Not Allowed
- 400 Bad Request
- 415 Unsupported Media Type

Runtime errors can result in a variety of exceptions in any @RestController methods. If these exceptions are not caught and handled appropriately, a generic 500 Internal Server Error is returned.

Restart the app

Make a GET request to:

```
localhost:8080/api/demo-error
```

Check the body of the response for the 500 Internal Server Error and the customized error message and verify the occurrence of the Divide by zero ArithmeticException on the server side.

6.1   Creating a custom exception

We can provide a custom Exception class of our own (ResourceNotFoundException) with a specific HTTP status code (404 Not Found) and error message (via the `reason` attribute of @ResponseStatus).

Make a GET request to:

`localhost:8080/api/firstdemo/10`

Note that a valid Resume object is returned.

Make a GET request to:

`localhost:8080/api/firstdemo/200`

Note that 404 Not Found error is returned with a custom error message.

6.2   Using ResponseStatusException

Another way of implementing a custom Exception is through the ResponseStatusException class.

Make a GET request to:

`localhost:8080/api/seconddemo/10`

Note that a valid Resume object is returned.

Make a GET request to:

`localhost:8080/api/seconddemo/200`

Note that 404 Not Found error is returned with a custom error message. This error message can be dynamically set up instead of being hardcoded as in the case of @ResponseStatus.

6.3   Creating @ControllerAdvice class with @ExceptionHandler methods

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

```
SampleControllerExceptionHandler
CustomErrorMessage
```

We can use the @ControllerAdvice to annotate a class that aggregates together specialized exception handling methods that (marked with @ExceptionHandler) that will catch and handle common Java exceptions (such as NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException). They can also catch and handle custom user-defined Exceptions and return custom error messages if necessary.

Make a GET request to:

`localhost:8080/api/demo-error`

Notice this time that no 500 Internal Server Error response is returned; instead the appropriate exception handler logic is executed in `SampleControllerExceptionHandler`

Make a GET request to:

```
localhost:8080/api/thirddemo/999
```

Notice that the error message returned has a customized format which includes additional fields that are not present in typical Spring framework generated messages.

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

```
application.properties-v2
```

This includes some additional properties to increase the amount of error information provided in the standard Spring framework error response. You can check for all the server error related properties by searching for `server.error` at:

https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#application-properties.server

Make a POST request to:

```
localhost:8080/api/fourthdemo
```

with this malformed JSON content:

```
{
    "name": Peter",
}
```

Notice that the error message returned is significantly more complex. Typically, we don't set those properties to feedback so much information in production grade apps as this can pose a security risk to hackers who are constantly probing error messages for vulnerabilities in the app. These properties should therefore be only included in the development process.

In `src/main/java`, make the following changes:

```
SampleControllerExceptionHandler-v2
```

Repeat the POST request to:

```
localhost:8080/api/fourthdemo
```

with the same malformed JSON content:

```
{
    "name": Peter",
}
```

Notice this time a custom error message is returned instead due to the exception being caught and handled in the @ControllerAdvice class.

We can use this approach to catch and handle other similar exceptions thrown by the Spring framework as shown at this link:

https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-ann-rest-spring-mvc-exceptions