INSTRUCTIONS

RESTful Applications with Spring Boot

Purpose

In this lab you will use some of the features of Spring that support RESTful web services. Note that there's more than we can cover in this lab, please refer back to the presentation for a good overview.

Learning Outcomes

What you will learn:

1. Working with RESTful URLs that expose resources
2. Mapping request and response bodies using HTTP message converters
3. Use Spring MVC to implement server-side REST
4. Writing a programmatic HTTP client to consume RESTful web services

Specific subjects you will gain experience with:

1. Processing URI Templates using `@PathVariable`
2. Using `@RequestBody` and `@ResponseBody`
3. Using the `RestTemplate`

You will be using the *38-rest-ws* project.

Estimated time to complete: 50 minutes.

Use Case

With the emergence of the Single Page Application (SPA) architecture and with the need to support diverse set of client types including mobile devices, more and more back-end applications are developed and deployed as RESTful web services. Spring makes building RESTful web services very easy with a rich set of annotations.

In the prior MVC lab, you implemented a few RESTful endpoints for obtaining a list of accounts and details for a specific account. In this lab, you will complete the implementation of the `AccountController` by adding RESTful methods to create a new account, add a beneficiary to an account and delete an account.

Quick Instructions

Quick instructions for this exercise have been embedded within the lab materials in the form of TODO comments.

Make sure to stop any account-server you have running from a previous lab.

Instructions

The instructions for this lab are organized into sections. In the first section, you'll add support for retrieving a JSON-representation of accounts and their beneficiaries and test that using the `RestTemplate`.

In the second section, you'll add support for making changes by adding an account and adding and removing a beneficiary.

The optional bonus section will let you map an existing exception to a specific HTTP status code.

Exposing accounts and beneficiaries as RESTful resources

In this section, you'll expose accounts and beneficiaries as RESTful resources using Spring's URI template support, HTTP Message Converters and the `RestTemplate`.

Inspect the current application

TODO-01 : Run the Spring Boot application

- Open the `RestWsApplication` class in the `accounts` package to see how the application is bootstrapped: it imports the `AppConfig` configuration class, which contains an `accountManager` bean that provides transactional data access operations to manage `Account` instances.

  As `RestWsApplication` uses the `@SpringBootApplication` annotation, it will use component scanning and will define a bean for the `AccountController` class.

- Find `AccountClientTests` JUnit test case under the `src/test/java` source folder: this is what you'll use to interact with the running RESTful web services on the server.
- Run this Spring Boot application and verify that the application deployed successfully by accessing your home page from a browser.

  When you see the welcome page, the application was started successfully. If you have a problem starting the application, the most likely cause is that you already have an application from a previous lab still running on the same port, so be sure to stop it.

Expose the list of accounts

TODO-02 : Review implementation for getting a list of accounts (`accountSummary`)

The prior lab had you implement the REST endpoint to obtain the list of accounts. Take a moment to review the code. Note the URI and the HTTP verb used to perform this action. Note also what is being returned.

- Access account from same browser.

  Depending on the browser used, you may see the response inline (Chrome) or you may see a popup asking you what to do with the response: save it to a local file and open that in a local text editor (IE). You'll see that you've just received a response using a JSON representation (JavaScript Object Notation). How is that possible?

  The reason is that the project includes the Jackson library on its classpath - check the dependencies of your project.

  If this is the case, an HTTP Message Converter that uses Jackson will be active by default. The library mostly *just works* with our classes without further configuration. If you are interested you can have a look at the `MonetaryAmount` and `Percentage` classes and search for the Json annotations to see the additional configuration.

Retrieve the list of accounts using a `RestTemplate`

TODO-03 : Test the `accountSummary()` REST endpoint for listing accounts

- A client can process this JSON response anyway it sees fit. In our case, we'll rely on the same HTTP Message Converter to deserialize the JSON contents back into `Account` objects.
- Open the `AccountClientTests` class under the `src/test/java` source folder in the `accounts.client` package.

  This class uses a plain `RestTemplate` to connect to the server. Use the supplied template to retrieve the list of accounts from the server, from the same URL that you used in your browser.

You can use the `BASE_URL` variable as a starting point to build the full URL to use.

  We cannot assign to a `List<Account>` here, since Jackson won't be able to determine the generic type to deserialize to in that case: therefore we use an `Account[]` instead.
- Run the test and make sure that the `listAccounts` test succeeds.

Expose a single account

TODO-04 : Review the REST endpoint for the `accountDetails()` method

This method has also been implemented in the prior lab. Examine the implementation and note the URI, HTTP verb and what is being returned.

- To test your code, just try to access account 0 from your browser to verify the result.

TODO-05 : Test the `accountDetails` REST endpoint

- When you are done with the controller, complete the `AccountClientTests` by retrieving the account with id 0.

  The `RestTemplate` also supports URI templates with variables, so use one and pass 0 as a the value for the `urlVariables` varargs parameter.

- Run the test and ensure that the `getAccount` test now succeeds as well.

If the time allocated for the lab is almost used up, this is a good place to stop.

Modifying data using POST and DELETE

In this section we will create and remove data. From here on, you will implement both the functionality in the `AccountController` class and the test code in the `AccountClientTests` class.

Create a new account

So far we've only exposed resources by responding to GET requests. Now you'll add support for creating a new account as a new resource.

TODO-06 : Make a REST endpoint from `createAccount` method

- Update the `createAccount` method by adding a mapping to it from POST request to `/accounts`.

  The body of the POST will contain a JSON representation of an `Account`, just like the representation that our client received in the previous step.

- Make sure to annotate the `account` parameter appropriately to let the request's body be un-marshaled into an `Account` object.

When the method completes successfully, the client should receive a `201 Created` instead of `200 OK`, so annotate the `createAccount` method to make that happen as well.

- We will write a test for this method in an upcoming step.

TODO-07 : Add `Location` response header

- RESTful clients that receive a `201 Created` response will expect a `Location` response header in the response containing the URL of the newly created resource.
- Complete the TODO by implementing `entityWithLocation()`.
- Save all work and restart the application.

  To help you generate the full URL on which the new account can be accessed, you will need to use a `ServletUriComponentsBuilder` - refer back to the example in the slides to see how. This approach means you can avoid hard-coding the server name and servlet mapping in your controller code! You should use a `ResponseEntity` to generate the response.

TODO-08 : Test the `createAccount` REST endpoint

- Complete the test method by POSTing the given `Account` to the `/accounts` URL.

  The `RestTemplate` has two methods for this. Use the one that returns the location of the newly created resource and assign that to a variable.

TODO-09 : Retrieve the new account on the given location

- The returned `Account` will be equal to the one you POSTed, but will also have been given an `entityId` when it was saved to the database.
- Run the tests again and see if the `createAccount` test runs successfully.
- Regardless of whether this is the case or not, proceed with the next step! (You will use a tool to debug a problem in case the test failed.)

Seeing what happens at the HTTP level

If your *create account* test worked, and you are running short of time, you may skip this section and move on to "Create and delete a beneficiary". However the HTTP monitor discussed in this section is a useful debugging tool to know.

OPTIONAL: Create and Delete a Beneficiary

This is a long lab. If you are running short of time, this is a good place to stop. Otherwise let's manipulate Beneficiaries.

TODO 10-11 : Make a REST endpoint from `addBeneficiary` method

- Complete the `addBeneficiary` method in the `AccountController`.

  This is similar to what you did in the previous step, but now you also have to use a URI template to parse the `accountId`.

  Note that the request's body will only contain the name of the beneficiary: an HTTP Message Converter that will convert this to a `String` is enabled by default, so simply annotate the method parameter again to obtain the name.

- Create a `ResponseEntity` containing the location of the newly created beneficiary.

  As you can see in the `getBeneficiary` method, the name of the beneficiary is used to identify it in the URL. You can use `entityWithLocation()` again for setting the location field.

- Save all work and restart the application.

TODO-12 : Make a REST endpoint from `removeBeneficiary` method

- Complete the `removeBeneficiary` method. This time, return a `204 No Content` status.

TODO 13-16 : Test the REST endpoints

- To test your work, switch to the `AccountClientTests` and complete the TODOs.
- When you are done, run the test and verify that this time all test methods run successfully.
- If this is the case, you've completed the lab!

OPTIONAL BONUS: Return a `409 Conflict` when creating an account with an Existing Number

TODO-17 : Add an exception handler for `409 Conflict`

- The current test ensures that we always create a new account using a unique number. Let's change that and see what happens.
- Edit the `createAccount` method in the `AccountClientTests` test class to use a fixed account number, like `"123123123"`.
- Run the test: the first time it should succeed.
- Run the test again: this time it should fail.
- When you look at the exception in the JUnit View or at the response in the TCP/IP monitor, you will see that the server returned a `500 Internal Server Error`.
- If you look in the Console View for the server, you will see what caused this: a `DataIntegrityViolationConstraint`, ultimately caused by a

`ConstraintViolationException` indicating that the account number is violating a unique constraint.

- This is not really a server error: this is caused by the client providing conflicting data when attempting to create a new account.

- To properly indicate that to the client, we should return a `409 Conflict` rather than the `500 Internal Server Error` that's returned by default for uncaught exceptions.

- To enable this, add a new exception handling method that returns the correct code in case of a `DataIntegrityViolationConstraint`.

  Have a look at the `handleNotFound` method in the `AccountConroller` class or in the Advanced section in the slides for a way to do this.

- When you're done, run the test again (do it twice as the database will re-initialize on redeploy) and check that you now receive the correct status code.

- Optionally you can even restore the test method and create a new test method that verifies the new behavior.

Summary

You have refactored "Rewards" Spring MVC application as RESTful Web service by implementing RESTful endpoints using Spring provided annotations. You also have built RESTful client leveraging `RestTemplate` utility class.