Project Assignment Solution Outline
Paynet REST API workshop Jul - Aug 2021

1 Project solution outline

This project solution is based on this public REST API from RapidAPI:

https://rapidapi.com/apilayernet/api/rest-countries-v1

The following structure is used to store information related to a country (a subset of the original schema used in the public REST API):

Country
`id: Integer`
`name: String`
`capital: String`
`region: String`
`subregion: String`
`population: Long`
`currencies: Array [String]`
`languages: Array [String]`

The following HTTP methods and REST API endpoints are implemented:

Method	**API endpoint**	**Purpose**
`GET`	`/api/countries`	To retrieve all countries
`GET`	`/api/countries/XXXX`	To retrieve a specific country by its name
`GET`	`/api/countries/capital/XXXX`	To retrieve a specific country by its capital name
`GET`	`/api/countries?region=XXX`	To retrieve countries in a specific region
`GET`	`/api/countries?subregion=XXX`	To retrieve countries in a specific subregion
`GET`	`/api/countries?currency=XXX`	To retrieve countries that use a specific currency
`GET`	`/api/countries?language=XXX`	To retrieve countries which use a specific language
`GET`	`/api/countries?langnum=XXX`	To retrieve countries which use a certain number of languages (1, 2 and so on).
`GET`	`/api/countries/count?region=XXX`	To count the number of countries in a specific region

GET	/api/countries/count?language=XXX	To count the number of countries which use a specific language
GET	/api/countries?pop>X | pop<X	To retrieve countries whose population is more than or less than a certain number
GET	/api/countries?sort=pop-asc | sort=pop-desc	To retrieve countries in sorted order (either ascending or descending order) based on their population size
GET	/api/countries?limit=X&page=Y	Pagination functionality to retrieve page no Y containing a total of X countries. This can be combined with all of the other query parameters specified earlier
POST	/api/countries	Add a new country
PUT	/api/countries/XXXX	Make a complete modification to a country with the name XXXX
PATCH	/api/countries/XXXX	Make a partial modification to a country with the name XXXX
DELETE	/api/countries/XXXX	Delete the country with the name XXXX

2 REST API Service implementation

2.1 Key details

CountryDTO is created with an appropriate structure to serialize/deserialize JSON content from/to the original public REST API as well as JSON content from/to the local client. The JSON content returned from the original REST API uses a String array for the `currencies` and `languages` keys. Since MySQL does not have a built-in data type for arrays, we represent each of these fields in the @Entity Country domain class as a single String with the original array items separated by commas. We provide constructors in both classes to facilitate conversion between both of them.

In addition to CountryDTO, we define two more DTO classes for the purposes of serializing / deserializing into JSON content to be returned / received from the client. CustomErrorMessage returns our custom error message, while TotalCountDTO returns the response structure related to processing the `GET /api/countries/count` mapping methods.

Typically, we would use a unique numeric id to identify a single resource to retrieve, modify, update or delete. However, since all countries here have a unique name, we can use the country name instead as this is more intuitive for a client to use. So, in all the URLs for GET, POST, PUT and DELETE: we would specify `XXX` as the country name: `/api/countries/XXXX`

In CountryRepository, we use a combination of native SQL queries and derived query methods to perform the various CRUD operations. We need to remember to set @Modifying and @Transaction on the query that performs an update operation.

CountryController provides mapping for all the standard HTTP REST methods, including PATCH. For the case of the POST mapping, we return status 200 OK with no content instead of putting a value in the Location header. This is because for the case of this API, it is understood that we can retrieve a newly added record by simply reusing the same name that was used in the JSON post in a subsequent GET call. There is no need for the use of an internally generated ID that is unknown to the client, and which must therefore be returned to the client to use.

We support PATCH functionality by using the JsonPatch class and marking the media type as `application/json-patch+json`. This means we also have to ensure that we set this media type explicitly in the Content-Type header of all client applications that utilize this particular method. https://www.baeldung.com/spring-rest-json-patch

We provide the necessary values for communicating with the RapidAPI host in order to initialize the database table via properties specified in `application.properties` that are subsequently used in initializeFromPublicAPI in CountryService. This uses the @Bean factory method in MainConfig to produce a suitable RestTemplate object to perform the communication with the RapidAPI host.

The vast majority of key business logic is focused in the getCountriesWithParams and getCountWithParams methods in CountryService, which process the various possible query parameters and implements the filtering functionality associated with each of them.

We provide 3 different kinds of user-defined exceptions which are handled appropriately in an @ExceptionHandler method in the main @ControllerAdvice class: CountryControllerExceptionHandler. These exceptions are self-explanatory based on their names.

We create a class ServerConstants to house all the String constants that we use in our business logic. These are mainly related to the key names of the various query parameters that can be used in the GET method calls.

2.2 Running

The project root folder is: `ProjectRESTService`. Import this as a Maven project into STS (or Eclipse Enterprise Java IDE).

Ensure the MySQL server is running at its default port of 3306 and the suitable initialization has been done for a user account in accordance to the values of the various `spring.datasource` properties in the project's `application.properties`. For e.g. you will need to create a user with a specified password and grant it full access privilege to the database `workshopdb`:

```
CREATE USER 'spiderman'@'%' IDENTIFIED BY 'peterparker';
GRANT ALL ON workshopdb.* TO 'spiderman'@'%';
```

We can check that the user account has been created successfully with:

```
SELECT user FROM mysql.user;
```

We can check the privileges granted to this account with:

```
SHOW GRANTS FOR 'spiderman'@'%';
```

In `application.properties`, provide the value for your RapidAPI key:

```
rapidapi.key=xxx
```

The application provides two ways to initialize the local database table with the contents of the public REST API when it starts up:

a) Send a GET to retrieve the entire list of countries from the public REST API and then populate the local database table with the content
b) Parse the contents of a local JSON file containing this entire list of countries and then populate the local database table with the content

For approach a), we will provide this property setting in `application.properties`:

```
initialize.mode=server
```

For approach b), the property setting will be:

```
initialize.mode=file
json.file=allcountries.json
```

Ensure that the JSON file is placed in `src/main/resources`.

Notice the setting for this property:

```
spring.jpa.hibernate.ddl-auto=create
```

means that the application will overwrite any existing table when it starts up. If you already have initialized a database table from the first run of this app, and wish to maintain its contents the next time you start up the app, then change the property to:

```
spring.jpa.hibernate.ddl-auto=update
```

and make sure you set the `initialize.mode` property to any random value other than `file` or `server`.

The app will run in the normal Spring Boot embedded Tomcat server at the default port of 8080. You can change this by configuring `server.port=xxx` in `application.properties`.

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

2.3 Testing

Use Postman for testing.

For retrieving all countries, we use: GET /api/countries

```
localhost:8080/api/countries
```

To retrieve a specific country by its name, we use GET /api/countries/XXXX

```
localhost:8080/api/countries/thailand
localhost:8080/api/countries/malaysia
localhost:8080/api/countries/united kingdom
```

To test for errors, do a GET to:

```
localhost:8080/api/countries/asdf
```

To retrieve a specific country by its capital name, we use GET /api/countries/capital/XXXX

```
localhost:8080/api/countries/capital/london
localhost:8080/api/countries/capital/bangkok
localhost:8080/api/countries/capital/kuala lumpur
```

To test for errors, do a GET to:

```
localhost:8080/api/countries/capital/asdf
```

To retrieve countries in a specific region, GET /api/countries?region=XXX

```
localhost:8080/api/countries?region=Asia
localhost:8080/api/countries?region=Europe
localhost:8080/api/countries?region=Africa
```

To retrieve countries in a specific subregion, GET /api/countries?subregion=XXX

```
localhost:8080/api/countries?subregion=Northern Europe
localhost:8080/api/countries?subregion=Southern Asia
localhost:8080/api/countries?subregion=Western Africa
```

To retrieve countries that use a specific currency, GET /api/countries?currency=XXX

```
localhost:8080/api/countries?currency=USD
localhost:8080/api/countries?currency=EUR
localhost:8080/api/countries?currency=GBP
```

To retrieve countries which use a specific language, GET /api/countries?language=XXX

```
localhost:8080/api/countries?language=en
localhost:8080/api/countries?language=fr
localhost:8080/api/countries?language=de
```

To retrieve countries which use a certain number of languages, GET /api/countries?langnum=XXX

```
localhost:8080/api/countries?langnum=3
localhost:8080/api/countries?langnum=4
```

To test for errors, do a GET to:

```
localhost:8080/api/countries?langnum=xxx
```

To count the number of countries in a specific region, GET /api/countries/count?region=XXX

```
localhost:8080/api/countries/count?region=Asia
localhost:8080/api/countries/count?region=Europe
localhost:8080/api/countries/count?region=Africa
```

To count the number of countries which use a specific language,
GET /api/countries/count?language=XXX

```
localhost:8080/api/countries/count?language=en
localhost:8080/api/countries/count?language=fr
localhost:8080/api/countries/count?language=de
```

To retrieve countries whose population is more than or less than a certain number,
GET /api/countries?pop>X | pop<X

```
localhost:8080/api/countries?pop>100000000
localhost:8080/api/countries?pop<3000
```

To test for errors, do a GET to:

```
localhost:8080/api/countries?pop<asdf
localhost:8080/api/countries?pop=3000
```

To retrieve countries in sorted order based on their population size, GET/api/countries?sort=XXX

```
localhost:8080/api/countries?sort=pop-asc
localhost:8080/api/countries?sort=pop-desc
```

To test errors, do a GET to:

```
localhost:8080/api/countries?sort=asdf
```

Pagination functionality to retrieve page no Y containing a total of X countries,
GET /api/countries?limit=X&page=Y

```
localhost:8080/api/countries?limit=10&page=1
localhost:8080/api/countries?limit=5&page=0&region=Asia
localhost:8080/api/countries?limit=10&page=2&language=en
localhost:8080/api/countries?limit=4&page=2&currency=USD
```

To test for errors, do a GET to:

```
localhost:8080/api/countries?limit=xx
localhost:8080/api/countries?page=yy
```

Add a new country, POST /api/countries

```
localhost:8080/api/countries
```

```
{
    "name": "Asgard",
```

```
    "capital": "Odin City",
    "region": "Asia",
    "subregion": "Southern Asia",
    "population": 500800,
    "currencies": [
        "EUR",
        "USD"
    ],
    "languages": [
        "en",
        "fr",
        "de"
    ]
}
```

Check again with GET to:

`localhost:8080/api/countries/Asgard`

To test for errors, either do a POST with malformed JSON or with one of the fields missing (e.g. name, capital, region, etc).

Make a complete modification to a country, PUT /api/countries/XXXX

`localhost:8080/api/countries/Thailand`

```
{
    "name": "Thailand",
    "capital": "Ayuthia",
    "region": "Europe",
    "subregion": "Northern Europe",
    "population": 22300700,
    "currencies": [
        "USD",
        "KHR"
    ],
    "languages": [
        "th",
        "en"
    ]
}
```

Check again with GET to:

`localhost:8080/api/countries/Thailand`

To test for errors, either do a POST with malformed JSON or with one of the fields missing (e.g. name, capital, region, etc).

Make a partial modification to a country, PATCH /api/countries/XXXX

Change the `Content-Type` header to `application/json-patch+json`
Then do a PATCH to:

`localhost:8080/api/countries/france`

```
[
    {"op":"replace","path":"/capital","value":"awesome place"},
    {"op":"replace","path":"/region","value":"Asia"},
    {"op":"replace","path":"/subregion","value":"Southern Asia"}

]
```

Check again with GET to:

`localhost:8080/api/countries/france`

To test for errors, either do a POST with malformed JSON or with one of the fields missing (e.g. name, capital, region, etc).

Delete a country, DELETE /api/countries/XXXX

`localhost:8080/api/countries/zambia`

Check again with GET to:

`localhost:8080/api/countries/zambia`

To test for errors, DELETE to:

`localhost:8080/api/countries/asdf`

3 REST client CLI implementation

3.1 Key details

This uses the Picocli library (https://picocli.info/). This is integrated into the Spring Boot project through an appropriate modification to the main @SpringBootApplication class ProjectRestClientApplication (https://picocli.info/#_spring_boot_example)

The classes implementing the commands / subcommands are placed in the `com.workshop.jpa.commands` package. CallAPI is the main command which is followed by the relevant subcommand class to initiate the appropriate REST API call. All of these classes declare relevant options or parameters and based on these values, will make a call to methods in

MyRestService which in turn makes the actual HTTP REST method calls using RestTemplate object. This class also has some methods to validate some arguments that are used as well as read JSON content from a file in order to place it in an outgoing request.

The Apache Commons HTTP Client is used as the underlying HTTP client for the RestTemplate object, which is configured in RestTemplateConfig. This is because the default HTTP client for RestTemplate, HttpURLConnection does not support the use of the PATCH HTTP method. This also needs to be included as a POM dependency:

```
<dependency>
	<groupId>org.apache.httpcomponents</groupId>
	<artifactId>httpclient</artifactId>
</dependency>
```

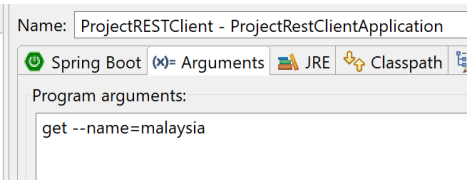
3.2 Running

The project root folder is: `ProjectCLIClient`. Import this as a Maven project into STS (or Eclipse Enterprise Java IDE).

Start up the REST API service. If you wish to use TCP/IP monitor in Eclipse to display HTTP traffic between the service and the client, make sure you configure the `myrest.url` property in `application.properties` to send initially to the monitor port and configure the monitor to redirect to the port that the service is listening on.

To run the client application, you will need to pass it the appropriate command line parameters / arguments.

From inside STS, you can do this by going to the Run Configuration for the project, and typing in the appropriate arguments, e.g.



You can also create an executable JAR by running the Maven build goals: `clean package`

Then in the command prompt / terminal in the directory containing the generated JAR, run it and pass the appropriate command line parameters / arguments, for e.g.

```
java -jar ProjectCLIClient.jar get --name=Malaysia
```

3.3 Testing

Use the following arguments:

To get general help for the subcommands and subcommand options:

```
help
help get
help put
help post
help patch
help delete
```

For retrieving all countries:

```
get --name=*
```

To retrieve a specific country by its name:

```
get --name=thailand
get --name=malaysia
get --name="united kingdom"
```

To test for errors:

```
get --name=asdf
```

To retrieve a specific country by its capital name

```
get --capital=london
get --capital=Bangkok
get --capital="kuala lumpur"
```

To test for errors:

```
get --capital=asdf
```

To retrieve countries in a specific region:

```
get --region=Asia
get --region=Europe
get --region=Africa
```

To retrieve countries in a specific subregion:

```
get --subregion=Northern Europe
get --subregion=Southern Asia
get --subregion=Western Africa
```

To retrieve countries that use a specific currency:

```
get --currency=USD
```

```
get --currency=EUR
get --currency=GBP
```

To retrieve countries which use a specific language:

```
get --language=en
get --language=fr
get --language=de
```

To retrieve countries which use a certain number of languages:

```
get --langnum=3
get --langnum=4
```

To test for errors:

```
get --langnum=xxx
```

To count the number of countries in a specific region:

```
get --count-region=Europe
get --count-region=Asia
get --count-region=Africa
```

To count the number of countries which use a specific language:

```
get --count-language=en
get --count-language=fr
get --count-language=de
```

To retrieve countries whose population is more than or less than a certain number:

```
get  --popmore=100000000
get  --popless=5000
```

To test for errors:

```
get  --popmore=asdf
```

To retrieve countries in sorted order based on their population size:

```
get  --sort=pop-asc
get  --sort=pop-desc
```

To test for errors:

```
get  --sort=asdf
```

Pagination functionality to retrieve page no Y containing a total of X countries:

```
get  --name=* --limit=10 --page=1
get  --region=Asia --limit=5 --page=0
get  --sort=pop-desc --limit=10 --page=0
```

To test for errors:

```
get  --name=* --limit=xx --page=1
get  --name=* --limit=10
```

For the POST, PUT and PATCH commands, the client app will retrieve the required JSON data to send with the request from a local file, which is specified as one of the arguments.

Some sample files are provided in the `data` folder.

Add a new country with POST

```
post post.json
```

Check again with GET to:

```
get --name=shangrila
```

Make a complete modification to a country with PUT

```
put put.json
```

Check again with GET to:

```
get --name=tuvalu
```

Make a partial modification to a country, PATCH /api/countries/XXXX

```
patch spain patch.json
```

Check again with GET to:

```
get --name=spain
```

Delete a country, DELETE /api/countries/XXXX

```
delete vanuatu
```

Check again with GET to:

```
get --name=vanuatu
```

To test for errors:

```
delete asdf
```

4 REST client Spring MVC implementation

4.1 Key details

This implementation uses Thymeleaf (a template engine that is a modern alternative to JSP) in Spring MVC for implementing the front-end web UI.
https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html
https://frontbackend.com/thymeleaf/thymeleaf-tutorial

All the webpages that the user interacts with are Thymeleaf templates (HTML files) stored in the `templates` folder of `src/main/resources`. There is a single @Controller class (FormsController) that implements handlers for returning and processing the various forms used.

Some of the main form templates (`add.html, update.html, modify.html`) use the Bean Validation API to validate the form input data and return appropriate validation error messages if invalid input is encountered. This requires adding the I/O -> Validation starter dependency when creating your project. There are two classes (CountryDetailsForm and GetCountryForm) which are used for object binding with the form data for this purpose. MessagesConfig is used to configure a file (`custom-validation-messages.properties`) to store custom validation messages for specific errors in specific form elements.

The MyService class originally used in the REST CLI implementation for implementing the various HTTP REST calls to the REST API service is reused here, but refactored slightly to reflect its use with a web form-based UI rather than a CLI. We also create two new exceptions (PageListParamException, GetParamException) to help in this integration.

4.2 Running

The web app that handles the GET and POST requests for the Thymeleaf templates will need to run on a Tomcat server that listens on a different port than the one for the REST API service. You can set this by configuring `server.port=xxx` in `application.properties`.

Start up the REST API service. If you wish to use TCP/IP monitor in Eclipse to display HTTP traffic between the service and the client, make sure you configure the `myrest.url` property in `application.properties` to send initially to the monitor port and configure the monitor to redirect to the port that the service is listening on.

Start up the web app in the usual manner from the Boot dashboard. Navigate to `localhost:xxx` to retrieve the main page of the app and navigate through it by clicking on the appropriate links.

4.3 Testing

The web pages are relatively intuitive to use and navigate through and you can use the examples from the Spring REST CLI implementation to verify the functionality of this Spring MVC app.